Solidity Documentation

Release 0.8.37-develop

Ethereum

Jul 28, 2026

BASICS

1 Getting Started **3**

2 Translations **5**

3 Contents **7**

3.1 Introduction to Smart Contracts . . . 7
3.2 Solidity by Example . . . 16
3.3 Installing the Solidity Compiler . . . 38
3.4 Layout of a Solidity Source File . . . 47
3.5 Structure of a Contract . . . 50
3.6 Types . . . 53
3.7 Units and Globally Available Variables . . . 92
3.8 Expressions and Control Structures . . . 100
3.9 Contracts . . . 114
3.10 Inline Assembly . . . 160
3.11 Cheatsheet . . . 166
3.12 Language Grammar . . . 170
3.13 Using the Compiler . . . 198
3.14 Analysing the Compiler Output . . . 215
3.15 Solidity IR-based Codegen Changes . . . 218
3.16 Layout of State Variables in Storage and Transient Storage . . . 223
3.17 Layout in Memory . . . 233
3.18 Layout of Call Data . . . 234
3.19 Cleaning Up Variables . . . 234
3.20 Source Mappings . . . 235
3.21 The Optimizer . . . 236
3.22 Contract Metadata . . . 256
3.23 Contract ABI Specification . . . 261
3.24 Security Considerations . . . 275
3.25 List of Known Bugs . . . 282
3.26 Solidity v0.5.0 Breaking Changes . . . 306
3.27 Solidity v0.6.0 Breaking Changes . . . 313
3.28 Solidity v0.7.0 Breaking Changes . . . 316
3.29 Solidity v0.8.0 Breaking Changes . . . 318
3.30 NatSpec Format . . . 321
3.31 SMTChecker and Formal Verification . . . 325
3.32 Yul . . . 342
3.33 Import Path Resolution . . . 363
3.34 Style Guide . . . 373
3.35 Common Patterns . . . 394

3.36 Resources 400
3.37 Contributing 403
3.38 Language Influences 412
3.39 Solidity Brand Guide 412

Index **415**

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs that govern the behavior of accounts within the Ethereum state.

Solidity is a curly-bracket language designed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python, and JavaScript. You can find more details about which languages Solidity has been inspired by in the *language influences* section.

Solidity is statically typed, supports inheritance, libraries, and complex user-defined types, among other features.

With Solidity, you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

When deploying contracts, you should use the latest released version of Solidity. Apart from exceptional cases, only the latest version receives security fixes. Furthermore, breaking changes, as well as new features, are introduced regularly. We currently use a 0.y.z version number to indicate this fast pace of change.

Warning

Solidity recently released the 0.8.x version that introduced a lot of breaking changes. Make sure you read *the full list*.

Ideas for improving Solidity or this documentation are always welcome, read our *contributors guide* for more details.

Hint

You can download this documentation as PDF, HTML or Epub by clicking on the versions flyout menu in the bottom-right corner and selecting the preferred download format.

CHAPTER

ONE

GETTING STARTED

1. Understand the Smart Contract Basics

If you are new to the concept of smart contracts, we recommend you to get started by digging into the "Introduction to Smart Contracts" section, which covers the following:

- *A simple example smart contract* written in Solidity.
- *Blockchain Basics*.
- *The Ethereum Virtual Machine*.

2. Get to Know Solidity

Once you are accustomed to the basics, we recommend you read the *"Solidity by Example"* and "Language Description" sections to understand the core concepts of the language.

3. Install the Solidity Compiler

There are various ways to install the Solidity compiler, simply choose your preferred option and follow the steps outlined on the *installation page*.

Hint

You can try out code examples directly in your browser with the Remix IDE. Remix is a web browser-based IDE that allows you to write, deploy and administer Solidity smart contracts, without the need to install Solidity locally.

Warning

As humans write software, it can have bugs. Therefore, you should follow established software development best practices when writing your smart contracts. This includes code review, testing, audits, and correctness proofs. Smart contract users are sometimes more confident with code than their authors, and blockchains and smart contracts have their own unique issues to watch out for, so before working on production code, make sure you read the *Security Considerations* section.

4. Learn More

If you want to learn more about building decentralized applications on Ethereum, the Ethereum Developer Resources can help you with further general documentation around Ethereum, and a wide selection of tutorials, tools, and development frameworks.

If you have any questions, you can try searching for answers or asking on the Ethereum StackExchange, or our Gitter channel.

CHAPTER

TWO

TRANSLATIONS

Community contributors help translate this documentation into several languages. Note that they have varying degrees of completeness and up-to-dateness. The English version stands as a reference.

You can switch between languages by clicking on the flyout menu in the bottom-right corner and selecting the preferred language.

- Chinese
- French
- Indonesian
- Japanese
- Korean
- Persian
- Russian
- Spanish
- Turkish

Note

We set up a GitHub organization and translation workflow to help streamline the community efforts. Please refer to the translation guide in the solidity-docs org for information on how to start a new language or contribute to the community translations.

CHAPTER

THREE

CONTENTS

Keyword Index, Search Page

3.1 Introduction to Smart Contracts

3.1.1 A Simple Smart Contract

Let us begin with a basic example that sets the value of a variable and exposes it for other contracts to access. It is fine if you do not understand everything right now, we will go into more details later.

Storage Example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

The first line tells you that the source code is licensed under the GPL version 3.0. Machine-readable license specifiers are important in a setting where publishing the source code is the default.

The next line specifies that the source code is written for Solidity version 0.4.16, or a newer version of the language up to, but not including version 0.9.0. This is to ensure that the contract is not compilable with a new (breaking) compiler version, where it could behave differently. *Pragmas* are common instructions for compilers about how to treat the source code (e.g. pragma once).

A contract in the sense of Solidity is a collection of code (its *functions*) and data (its *state*) that resides at a specific address on the Ethereum blockchain. The line `uint storedData;` declares a state variable called `storedData` of type `uint` (*u*nsigned *int*eger of *256* bits). You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. In this example, the contract defines the functions `set` and `get` that can be used to modify or retrieve the value of the variable.

To access a member (like a state variable) of the current contract, you do not typically add the `this.` prefix, you just access it directly via its name. Unlike in some other languages, omitting it is not just a matter of style, it results in a

completely different way to access the member, but more on this later.

This contract does not do much yet apart from (due to the infrastructure built by Ethereum) allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number. Anyone could call `set` again with a different value and overwrite your number, but the number is still stored in the history of the blockchain. Later, you will see how you can impose access restrictions so that only you can alter the number.

Warning

Be careful with using Unicode text, as similar looking (or even identical) characters can have different code points and as such are encoded as a different byte array.

Note

All identifiers (contract names, function names and variable names) are restricted to the ASCII character set. It is possible to store UTF-8 encoded data in string variables.

Subcurrency Example

The following contract implements the simplest form of a cryptocurrency. The contract allows only its creator to create new coins (different issuance schemes are possible). Anyone can send coins to each other without a need for registering with a username and password, all you need is an Ethereum keypair.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.26;

// This will only compile via IR
contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping(address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }
```

(continues on next page)

(continued from previous page)

```
    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], InsufficientBalance(amount, balances[msg.
↪sender]));
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

This contract introduces some new concepts, let us go through them one by one.

The line `address public minter;` declares a state variable of type *address*. The `address` type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts, or a hash of the public half of a keypair belonging to *externally-owned accounts*.

The keyword `public` automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. The code of the function generated by the compiler is equivalent to the following (ignore `external` and `view` for now):

```
function minter() external view returns (address) { return minter; }
```

You could add a function like the above yourself, but you would have a function and state variable with the same name. You do not need to do this, the compiler figures it out for you.

The next line, `mapping(address => uint) public balances;` also creates a public state variable, but it is a more complex datatype. The *mapping* type maps addresses to *unsigned integers*.

Mappings can be seen as hash tables which are virtually initialized such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. Record what you added to the mapping, or use it in a context where this is not needed. Or even better, keep a list, or use a more suitable data type.

The *getter function* created by the `public` keyword is more complex in the case of a mapping. It looks like the following:

```
function balances(address account) external view returns (uint) {
    return balances[account];
}
```

You can use this function to query the balance of a single account.

The line `event Sent(address from, address to, uint amount);` declares an *"event"*, which is emitted in the last line of the function `send`. Ethereum clients such as web applications can listen for these events emitted on the blockchain without much cost. As soon as it is emitted, the listener receives the arguments `from`, `to` and `amount`, which makes it possible to track transactions.

To listen for this event, you could use the following JavaScript code, which uses web3.js to create the `Coin` contract object, and any user interface calls the automatically generated `balances` function from above:

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

The *constructor* is a special function that is executed during the creation of the contract and cannot be called afterwards. In this case, it permanently stores the address of the person creating the contract. The `msg` variable (together with `tx` and `block`) is a *special global variable* that contains properties which allow access to the blockchain. `msg.sender` is always the address where the current (external) function call came from.

The functions that make up the contract, and that users and contracts can call are `mint` and `send`.

The `mint` function sends an amount of newly created coins to another address. The *require* function call defines conditions that reverts all changes if not met. In this example, `require(msg.sender == minter);` ensures that only the creator of the contract can call `mint`. In general, the creator can mint as many tokens as they like, but at some point, this will lead to a phenomenon called "overflow". Note that because of the default *Checked arithmetic*, the transaction would revert if the expression `balances[receiver] += amount;` overflows, i.e., when `balances[receiver] + amount` in arbitrary precision arithmetic is larger than the maximum value of `uint` (`2**256 - 1`). This is also true for the statement `balances[receiver] += amount;` in the function `send`.

Errors allow you to provide more information to the caller about why a condition or operation failed. Errors are used together with the *revert statement*. The `revert` statement unconditionally aborts and reverts all changes, much like the *require function*. Both approaches allow you to provide the name of an error and additional data which will be supplied to the caller (and eventually to the front-end application or block explorer) so that a failure can more easily be debugged or reacted upon.

The `send` function can be used by anyone (who already has some of these coins) to send coins to anyone else. If the sender does *not* have enough coins to send, the condition in `require` evaluates to false, triggering a `revert` with the `InsufficientBalance` error. This error supplies the requested amount and available balance to the caller, which front-end applications or block explorers can surface for debugging.

Note

If you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the record that you sent coins and the changed balances are only stored in the data storage of this particular coin contract. By using events, you can create a "blockchain explorer" that tracks transactions and balances of your new coin, but you have to inspect the coin contract address and not the addresses of the coin owners.

3.1.2 Blockchain Basics

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) are just there to provide a certain set of features and promises for the platform. Once you accept these features as given, you do not have to worry about the underlying technology - or do you have to know how Amazon's AWS works internally in order to use it?

Transactions

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make (assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is being applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer some compensation, e.g. Ether, from it.

Blocks

One major obstacle to overcome is what (in Bitcoin terms) is called a "double-spend attack": What happens if two transactions exist in the network that both want to empty an account? Only one of the transactions can be valid, typically the one that is accepted first. The problem is that "first" is not an objective term in a peer-to-peer network.

The abstract answer to this is that you do not have to care. A globally accepted order of the transactions will be selected for you, solving the conflict. The transactions will be bundled into what is called a "block" and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time, and that is where the word "blockchain" derives from. Blocks are added to the chain at regular intervals, although these intervals may be subject to change in the future. For the most up-to-date information, it is recommended to monitor the network, for example, on Etherscan.

As part of the "order selection mechanism", which is called attestation, it may happen that blocks are reverted from time to time, but only at the "tip" of the chain. The more blocks are added on top of a particular block, the less likely this block will be reverted. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

Note

Transactions are not guaranteed to be included in the next block or any specific future block, since it is not up to the submitter of a transaction, but up to the miners to determine in which block the transaction is included.

If you want to schedule future calls of your contract, you can use a smart contract automation tool or an oracle service.

3.1.3 The Ethereum Virtual Machine

Overview

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

Accounts

There are two kinds of accounts in Ethereum which share the same address space: **Externally-owned accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an externally-owned account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called "nonce").

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in "Wei" to be exact, `1 ether` is `10**18 wei`) which can be modified by sending transactions that include Ether.

Transactions

A transaction is a message that is sent from one account to another account (which might be the same or empty, see below). It can include binary data (which is called "payload") and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is not set (the transaction does not have a recipient or the recipient is set to `null`), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the "nonce"). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

> **Note**
>
> While a contract is being created, its code is still empty. Because of that, you should not call back into the contract under construction until its constructor has finished executing.

Gas

Upon creation, each transaction is charged with a certain amount of **gas** that has to be paid for by the originator of the transaction (`tx.origin`). While the EVM executes the transaction, the gas is gradually depleted according to specific rules. If the gas is used up at any point (i.e. it would be negative), an out-of-gas exception is triggered, which ends execution and reverts all modifications made to the state in the current call frame.

This mechanism incentivizes economical use of EVM execution time and also compensates EVM executors (i.e. miners / stakers) for their work. Since each block has a maximum amount of gas, it also limits the amount of work needed to validate a block.

The **gas price** is a value set by the originator of the transaction, who has to pay `gas_price * gas` up front to the EVM executor. If some gas is left after execution, it is refunded to the transaction originator. In case of an exception that reverts changes, already used up gas is not refunded.

Since EVM executors can choose to include a transaction or not, transaction senders cannot abuse the system by setting a low gas price.

Storage, Transient Storage, Memory and the Stack

The Ethereum Virtual Machine has different areas where it can store data with the most prominent being storage, transient storage, memory and the stack.

Each account has a data area called **storage**, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract, it is comparatively costly to read, and even more to initialise and modify storage. Because of this cost, you should minimize what you store in persistent storage to what the contract needs to run. Store data like derived calculations, caching, and aggregates outside of the contract. A contract can neither read nor write to any storage apart from its own.

Similar to storage, there is another data area called **transient storage**, where the main difference is that it is reset at the end of each transaction. The values stored in this data location persist only across function calls originating from the first call of the transaction. When the transaction ends, the transient storage is reset and the values stored there become unavailable to calls in subsequent transactions. Despite this, the cost of reading and writing to transient storage is significantly lower than for storage.

The third data area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory in order to get deeper access to the stack, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

Calldata, Returndata and Code

There are also other data areas which are not as apparent as those discussed previously. However, they are routinely used during the execution of smart contract transactions.

The calldata region is the data sent to a transaction as part of a smart contract transaction. For example, when creating a contract, calldata would be the constructor code of the new contract. The parameters of external functions are always initially stored in calldata in an ABI-encoded form and only then decoded into the location specified in their declaration. If declared as `memory`, the compiler will eagerly decode them into memory at the beginning of the function, while marking them as `calldata` means that this will be done lazily, only when accessed. Value types and `storage` pointers are decoded directly onto the stack.

The returndata is the way a smart contract can return a value after a call. In general, external Solidity functions use the `return` keyword to ABI-encode values into the returndata area.

The code is the region where the EVM instructions of a smart contract are stored. Code is the bytes read, interpreted, and executed by the EVM during smart contract execution. Instruction data stored in the code is persistent as part of a contract account state field. Immutable and constant variables are stored in the code region. All references to immutables are replaced with the values assigned to them. A similar process is performed for constants which have their expressions inlined in the places where they are referenced in the smart contract code.

Instruction Set

The instruction set of the EVM is kept minimal in order to avoid incorrect or inconsistent implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words or on slices of memory (or other byte arrays). The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

For a complete list, please see the *list of opcodes* as part of the inline assembly documentation.

Message Calls

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions "bubble up" the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it has finished execution, it can return data which will be stored at a location in the caller's memory preallocated by the caller. All such calls are fully synchronous.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls. Furthermore, only 63/64th of the gas can be forwarded in a message call, which causes a depth limit of a little less than 1000 in practice.

Delegatecall and Libraries

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context (i.e. at the address) of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the "library" feature in Solidity: Reusable library code that can be applied to a contract's storage, e.g. in order to implement a complex data structure.

Logs

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement *events*. Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in bloom filters, it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain (so-called "light clients") can still find these logs.

Create

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address as a transaction would). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

Deactivate and Self-destruct

The only way to remove code from the blockchain is when a contract at that address performs the `selfdestruct` operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

Warning

From `EVM >= Cancun` onwards, `selfdestruct` will **only** send all Ether in the account to the given recipient and not destroy the contract. However, when `selfdestruct` is called in the same transaction that creates the contract calling it, the behaviour of `selfdestruct` before Cancun hardfork (i.e., `EVM <= Shanghai`) is preserved and will destroy the current contract, deleting any data, including storage keys, code and the account itself. See EIP-6780 for more details.

The new behaviour is the result of a network-wide change that affects all contracts present on the Ethereum mainnet and testnets. It is important to note that this change is dependent on the EVM version of the chain on which the contract is deployed. The `--evm-version` setting used when compiling the contract has no bearing on it.

Also, note that the `selfdestruct` opcode has been deprecated in Solidity version 0.8.18, as recommended by EIP-6049. The deprecation is still in effect and the compiler will still emit warnings on its use. Any use in newly deployed contracts is strongly discouraged even if the new behavior is taken into account. Future changes to the EVM might further reduce the functionality of the opcode.

Warning

Even if a contract is removed by `selfdestruct`, it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using `selfdestruct` is not the same as deleting data from a hard disk.

Note

Even if a contract's code does not contain a call to `selfdestruct`, it can still perform that operation using `delegatecall` or `callcode`.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

Precompiled Contracts

There is a small set of contract addresses that are special: The address range between 1 and (including) `0x0a` contains "precompiled contracts" that can be called as any other contract but their behavior (and their gas consumption) is not defined by EVM code stored at that address (they do not contain code) but instead is implemented in the EVM execution environment itself.

Different EVM-compatible chains might use a different set of precompiled contracts. It might also be possible that new precompiled contracts are added to the Ethereum main chain in the future, but you can reasonably expect them to always be in the range between 1 and `0xffff` (inclusive).

3.2 Solidity by Example

3.2.1 Voting

The following contract is quite complex, but showcases a lot of Solidity's features. It implements a voting contract. Of course, the main problems of electronic voting is how to assign voting rights to the correct persons and how to prevent manipulation. We will not solve all problems here, but at least we will show how delegated voting can be done so that vote counting is **automatic and completely transparent** at the same time.

The idea is to create one contract per ballot, providing a short name for each option. Then the creator of the contract who serves as chairperson will give the right to vote to each address individually.

The persons behind the addresses can then choose to either vote themselves or to delegate their vote to a person they trust.

At the end of the voting time, `winningProposal()` will return the proposal with the largest number of votes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
```

(continues on next page)

(continued from previous page)

```
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) external {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) external {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "You have no right to vote");
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
```

(continues on next page)

(continued from previous page)

```
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in delegation.");
        }

        Voter storage delegate_ = voters[to];

        // Voters cannot delegate to accounts that cannot vote.
        require(delegate_.weight >= 1);

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender]`.
        sender.voted = true;
        sender.delegate = to;

        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
```

(continues on next page)

(continued from previous page)

```
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() external view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Possible Improvements

Currently, many transactions are needed to assign the rights to vote to all participants. Moreover, if two or more proposals have the same number of votes, `winningProposal()` is not able to register a tie. Can you think of a way to fix these issues?

3.2.2 Blind Auction

In this section, we will show how easy it is to create a completely blind auction contract on Ethereum. We will start with an open auction where everyone can see the bids that are made and then extend this contract into a blind auction where it is not possible to see the actual bid until the bidding period ends.

Simple Open Auction

The general idea of the following simple auction contract is that everyone can send their bids during a bidding period. The bids already include sending some compensation, e.g. Ether, in order to bind the bidders to their bid. If the highest bid is raised, the previous highest bidder gets their Ether back. After the end of the bidding period, the contract has to be called manually for the beneficiary to receive their Ether - contracts cannot activate themselves.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
```

(continues on next page)

(continued from previous page)

```
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Errors that describe failures.

    // The triple-slash comments are so-called natspec
    // comments. They will be shown when the user
    // is asked to confirm a transaction or
    // when an error is displayed.

    /// The auction has already ended.
    error AuctionAlreadyEnded();
    /// There is already a higher or equal bid.
    error BidNotHighEnough(uint highestBid);
    /// The auction has not ended yet.
    error AuctionNotYetEnded();
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    /// Create a simple auction with `biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `beneficiaryAddress`.
    constructor(
        uint biddingTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        auctionEndTime = block.timestamp + biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() external payable {
        // No arguments are necessary, all
        // information is already part of
        // the transaction. The keyword payable
        // is required for the function to
        // be able to receive Ether.

        // Revert the call if the bidding
        // period is over.
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();

        // If the bid is not higher, send the
        // Ether back (the revert statement
        // will revert all changes in this
```

(continues on next page)

(continued from previous page)

```
        // function execution including
        // it having received the Ether).
        if (msg.value <= highestBid)
            revert BidNotHighEnough(highestBid);

        if (highestBid != 0) {
            // Sending back the Ether by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it could execute an untrusted contract.
            // It is always safer to let the recipients
            // withdraw their Ether themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `call` returns.
            pendingReturns[msg.sender] = 0;

            // msg.sender is not of type `address payable` and must be
            // explicitly converted using `payable(msg.sender)` in order
            // use the member function `call()`.
            (bool success, ) = payable(msg.sender).call{value: amount}("");
            if (!success) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() external {
        // It is a good guideline to structure functions that interact
        // with other contracts (i.e. they call functions or send Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could call
        // back into the current contract and modify the state or cause
        // effects (ether payout) to be performed multiple times.
```

(continues on next page)

(continued from previous page)

```
        // If functions called internally include interaction with external
        // contracts, they also have to be considered interaction with
        // external contracts.

        // 1. Conditions
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
        if (ended)
            revert AuctionEndAlreadyCalled();

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        (bool success, ) = beneficiary.call{value: highestBid}("");
        require(success);
    }
}
```

Blind Auction

The previous open auction is extended to a blind auction in the following. The advantage of a blind auction is that there is no time pressure towards the end of the bidding period. Creating a blind auction on a transparent computing platform might sound like a contradiction, but cryptography comes to the rescue.

During the **bidding period**, a bidder does not actually send their bid, but only a hashed version of it. Since it is currently considered practically impossible to find two (sufficiently long) values whose hash values are equal, the bidder commits to the bid by that. After the end of the bidding period, the bidders have to reveal their bids: They send their values unencrypted, and the contract checks that the hash value is the same as the one provided during the bidding period.

Another challenge is how to make the auction **binding and blind** at the same time: The only way to prevent the bidder from just not sending the Ether after they won the auction is to make them send it together with the bid. Since value transfers cannot be blinded in Ethereum, anyone can see the value.

The following contract solves this problem by accepting any value that is larger than the highest bid. Since this can of course only be checked during the reveal phase, some bids might be **invalid**, and this is on purpose (it even provides an explicit flag to place invalid bids with high-value transfers): Bidders can confuse competition by placing several high or low invalid bids.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;
```

(continues on next page)

(continued from previous page)

```
    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    // Errors that describe failures.

    /// The function has been called too early.
    /// Try again at `time`.
    error TooEarly(uint time);
    /// The function has been called too late.
    /// It cannot be called after `time`.
    error TooLate(uint time);
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    // Modifiers are a convenient way to validate inputs to
    // functions. `onlyBefore` is applied to `bid` below:
    // The new function body is the modifier's body where
    // `_` is replaced by the old function body.
    modifier onlyBefore(uint time) {
        if (block.timestamp >= time) revert TooLate(time);
        _;
    }
    modifier onlyAfter(uint time) {
        if (block.timestamp <= time) revert TooEarly(time);
        _;
    }

    constructor(
        uint biddingTime,
        uint revealTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        biddingEnd = block.timestamp + biddingTime;
        revealEnd = biddingEnd + revealTime;
    }

    /// Place a blinded bid with `blindedBid` =
    /// keccak256(abi.encodePacked(value, fake, secret)).
    /// The sent ether is only refunded if the bid is correctly
    /// revealed in the revealing phase. The bid is valid if the
    /// ether sent together with the bid is at least "value" and
    /// "fake" is not true. Setting "fake" to true and sending
    /// not the exact amount are ways to hide the real bid but
    /// still make the required deposit. The same address can
```

(continues on next page)

(continued from previous page)

```
    /// place multiple bids.
    function bid(bytes32 blindedBid)
        external
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: blindedBid,
            deposit: msg.value
        }));
    }

    /// Reveal your blinded bids. You will get a refund for all
    /// correctly blinded invalid bids and for all bids except for
    /// the totally highest.
    function reveal(
        uint[] calldata values,
        bool[] calldata fakes,
        bytes32[] calldata secrets
    )
        external
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(values.length == length);
        require(fakes.length == length);
        require(secrets.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (values[i], fakes[i], secrets[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bidToCheck.blindedBid = bytes32(0);
        }
        (bool success, ) = payable(msg.sender).call{value: refund}("");
        require(success);
    }
```

(continues on next page)

(continued from previous page)

```
    /// Withdraw a bid that was overbid.
    function withdraw() external {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `call` returns (see the remark above about
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

            (bool success, ) = payable(msg.sender).call{value: amount}("");
            require(success);
        }
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd()
        external
        onlyAfter(revealEnd)
    {
        if (ended) revert AuctionEndAlreadyCalled();
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        (bool success, ) = beneficiary.call{value: highestBid}("");
        require(success);
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }
}
```

3.2.3 Safe Remote Purchase

Purchasing goods remotely currently requires multiple parties that need to trust each other. The simplest configuration involves a seller and a buyer. The buyer would like to receive an item from the seller and the seller would like to get some compensation, e.g. Ether, in return. The problematic part is the shipment here: There is no way to determine for sure that the item arrived at the buyer.

There are multiple ways to solve this problem, but all fall short in one or the other way. In the following example, both parties have to put twice the value of the item into the contract as escrow. As soon as this happened, the Ether will stay locked inside the contract until the buyer confirms that they received the item. After that, the buyer is returned the value (half of their deposit) and the seller gets three times the value (their deposit plus the value). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their Ether is locked forever.

This contract of course does not solve the problem, but gives an overview of how you can use state machine-like constructs inside a contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // The state variable has a default value of the first member, `State.created`
    State public state;

    modifier condition(bool condition_) {
        require(condition_);
        _;
    }

    /// Only the buyer can call this function.
    error OnlyBuyer();
    /// Only the seller can call this function.
    error OnlySeller();
    /// The function cannot be called at the current state.
    error InvalidState();
    /// The provided value has to be even.
    error ValueNotEven();

    modifier onlyBuyer() {
        if (msg.sender != buyer)
            revert OnlyBuyer();
        _;
    }

    modifier onlySeller() {
        if (msg.sender != seller)
            revert OnlySeller();
        _;
    }

    modifier inState(State state_) {
        if (state != state_)
```

(continues on next page)

(continued from previous page)

```
            revert InvalidState();
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();
    event SellerRefunded();

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() payable {
        seller = payable(msg.sender);
        value = msg.value / 2;
        if ((2 * value) != msg.value)
            revert ValueNotEven();
    }

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        external
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        // We use call here directly. It is
        // reentrancy-safe, because it is the
        // last call in this function and we
        // already changed the state.
        (bool success, ) = seller.call{value: address(this).balance}("");
        require(success);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        external
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = payable(msg.sender);
        state = State.Locked;
    }
```

(continues on next page)

(continued from previous page)

```
    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        external
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `call` below
        // can call in again here.
        state = State.Release;

        (bool success, ) = buyer.call{value: value}("");
        require(success);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        external
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // It is important to change the state first because
        // otherwise, the contracts called using `call` below
        // can call in again here.
        state = State.Inactive;

        (bool success, ) = seller.call{value: 3 * value}("");
        require(success);
    }
}
```

3.2.4 Micropayment Channel

In this section, we will learn how to build an example implementation of a payment channel. It uses cryptographic signatures to make repeated transfers of Ether between the same parties secure, instantaneous, and without transaction fees. For the example, we need to understand how to sign and verify signatures, and setup the payment channel.

Creating and verifying signatures

Imagine Alice wants to send some Ether to Bob, i.e. Alice is the sender and Bob is the recipient.

Alice only needs to send cryptographically signed messages off-chain (e.g. via email) to Bob and it is similar to writing checks.

Alice and Bob use signatures to authorize transactions, which is possible with smart contracts on Ethereum. Alice will build a simple smart contract that lets her transmit Ether, but instead of calling a function herself to initiate a payment, she will let Bob do that, and therefore pay the transaction fee.

The contract will work as follows:

1. Alice deploys the `ReceiverPays` contract, attaching enough Ether to cover the payments that will be made.

2. Alice authorizes a payment by signing a message with her private key.
3. Alice sends the cryptographically signed message to Bob. The message does not need to be kept secret (explained later), and the mechanism for sending it does not matter.
4. Bob claims his payment by presenting the signed message to the smart contract, it verifies the authenticity of the message and then releases the funds.

Creating the signature

Alice does not need to interact with the Ethereum network to sign the transaction, the process is completely offline. In this tutorial, we will sign messages in the browser using web3.js and MetaMask, using the method described in EIP-712, as it provides a number of other security benefits.

```
/// Hashing first makes things easier
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log("Signed
↪"); });
```

Note

The `web3.eth.personal.sign` prepends the length of the message to the signed data. Since we hash first, the message will always be exactly 32 bytes long, and thus this length prefix is always the same.

What to Sign

For a contract that fulfills payments, the signed message must include:

1. The recipient's address.
2. The amount to be transferred.
3. Protection against replay attacks.

A replay attack is when a signed message is reused to claim authorization for a second action. To avoid replay attacks we use the same technique as in Ethereum transactions themselves, a so-called nonce, which is the number of transactions sent by an account. The smart contract checks if a nonce is used multiple times.

Another type of replay attack can occur when the owner deploys a `ReceiverPays` smart contract, makes some payments, and then destroys the contract. Later, they decide to deploy the `RecipientPays` smart contract again, but the new contract does not know the nonces used in the previous deployment, so the attacker can use the old messages again.

Alice can protect against this attack by including the contract's address in the message, and only messages containing the contract's address itself will be accepted. You can find an example of this in the first two lines of the `claimPayment()` function of the full contract at the end of this section.

Furthermore, instead of destroying the contract by calling `selfdestruct`, which is currently deprecated, we will disable the contract's functionalities by freezing it, resulting in the reversion of any call after it being frozen.

Packing arguments

Now that we have identified what information to include in the signed message, we are ready to put the message together, hash it, and sign it. For simplicity, we concatenate the data. The ethereumjs-abi library provides a function called `soliditySHA3` that mimics the behavior of Solidity's `keccak256` function applied to arguments encoded using `abi.encodePacked`. Here is a JavaScript function that creates the proper signature for the `ReceiverPays` example:

```
// recipient is the address that should be paid.
// amount, in wei, specifies how much ether should be sent.
// nonce can be any unique number to prevent replay attacks
// contractAddress is used to prevent cross-contract replay attacks
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

Recovering the Message Signer in Solidity

In general, ECDSA signatures consist of two parameters, `r` and `s`. Signatures in Ethereum include a third parameter called `v`, that you can use to verify which account's private key was used to sign the message, and the transaction's sender. Solidity provides a built-in function *ecrecover* that accepts a message along with the `r`, `s` and `v` parameters and returns the address that was used to sign the message.

Extracting the Signature Parameters

Signatures produced by web3.js are the concatenation of `r`, `s` and `v`, so the first step is to split these parameters apart. You can do this on the client-side, but doing it inside the smart contract means you only need to send one signature parameter rather than three. Splitting apart a byte array into its constituent parts is a mess, so we use *inline assembly* to do the job in the `splitSignature` function (the third function in the full contract at the end of this section).

Computing the Message Hash

The smart contract needs to know exactly what parameters were signed, and so it must recreate the message from the parameters and use that for signature verification. The functions `prefixed` and `recoverSigner` do this in the `claimPayment` function.

The full contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Owned {
    address payable owner;
    constructor() {
        owner = payable(msg.sender);
    }
}

contract Freezable is Owned {
    bool private _frozen = false;

    modifier notFrozen() {
        require(!_frozen, "Inactive Contract.");
        _;
    }
```

(continues on next page)

(continued from previous page)

```
    function freeze() internal {
        if (msg.sender == owner)
            _frozen = true;
    }
}

contract ReceiverPays is Freezable {
    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)
        external
        notFrozen
    {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // this recreates the message that was signed on the client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce,␣
↪this)));
        require(recoverSigner(message, signature) == owner);
        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success);
    }

    /// freeze the contract and reclaim the leftover funds.
    function shutdown()
        external
        notFrozen
    {
        require(msg.sender == owner);
        freeze();
        (bool success, ) = payable(msg.sender).call{value: address(this).balance}("");
        require(success);
    }

    /// signature methods.
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
            // second 32 bytes.
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes).
```

(continues on next page)

(continued from previous page)

```
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);
        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Writing a Simple Payment Channel

Alice now builds a simple but complete implementation of a payment channel. Payment channels use cryptographic signatures to make repeated transfers of Ether securely, instantaneously, and without transaction fees.

What is a Payment Channel?

Payment channels allow participants to make repeated transfers of Ether without using transactions. This means that you can avoid the delays and fees associated with transactions. We are going to explore a simple unidirectional payment channel between two parties (Alice and Bob). It involves three steps:

1. Alice funds a smart contract with Ether. This "opens" the payment channel.
2. Alice signs messages that specify how much of that Ether is owed to the recipient. This step is repeated for each payment.
3. Bob "closes" the payment channel, withdrawing his portion of the Ether and sending the remainder back to the sender.

Note

Only steps 1 and 3 require Ethereum transactions, step 2 means that the sender transmits a cryptographically signed message to the recipient via off chain methods (e.g. email). This means only two transactions are required to support any number of transfers.

Bob is guaranteed to receive his funds because the smart contract escrows the Ether and honours a valid signed message. The smart contract also enforces a timeout, so Alice is guaranteed to eventually recover her funds even if the recipient refuses to close the channel. It is up to the participants in a payment channel to decide how long to keep it open. For a short-lived transaction, such as paying an internet café for each minute of network access, the payment channel may be kept open for a limited duration. On the other hand, for a recurring payment, such as paying an employee an hourly wage, the payment channel may be kept open for several months or years.

Opening the Payment Channel

To open the payment channel, Alice deploys the smart contract, attaching the Ether to be escrowed and specifying the intended recipient and a maximum duration for the channel to exist. This is the `constructor` in the `SimplePaymentChannel` contract, at the end of this section.

Making Payments

Alice makes payments by sending signed messages to Bob. This step is performed entirely outside of the Ethereum network. Messages are cryptographically signed by the sender and then transmitted directly to the recipient.

Each message includes the following information:

- The smart contract's address, used to prevent cross-contract replay attacks.
- The total amount of Ether that is owed to the recipient so far.

A payment channel is closed just once, at the end of a series of transfers. Because of this, only one of the messages sent is redeemed. This is why each message specifies a cumulative total amount of Ether owed, rather than the amount of the individual micropayment. The recipient will naturally choose to redeem the most recent message because that is the one with the highest total. The nonce per-message is not needed anymore, because the smart contract only honours a single message. The address of the smart contract is still used to prevent a message intended for one payment channel from being used for a different channel.

Here is the modified JavaScript code to cryptographically sign a message from the previous section:

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.solidity SHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress is used to prevent cross-contract replay attacks.
// amount, in wei, specifies how much Ether should be sent.

function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
    signMessage(message, callback);
}
```

Closing the Payment Channel

When Bob is ready to receive his funds, it is time to close the payment channel by calling a `close` function on the smart contract. Closing the channel pays the recipient the Ether they are owed and deactivates the contract by freezing it, sending any remaining Ether back to Alice. To close the channel, Bob needs to provide a message signed by Alice.

The smart contract must verify that the message contains a valid signature from the sender. The process for doing this verification is the same as the process the recipient uses. The Solidity functions `isValidSignature` and

`recoverSigner` work just like their JavaScript counterparts in the previous section, with the latter function borrowed from the `ReceiverPays` contract.

Only the payment channel recipient can call the `close` function, who naturally passes the most recent payment message because that message carries the highest total owed. If the sender were allowed to call this function, they could provide a message with a lower amount and cheat the recipient out of what they are owed.

The function verifies the signed message matches the given parameters. If everything checks out, the recipient is sent their portion of the Ether, and the sender is sent the remaining funds via a `transfer`. You can see the `close` function in the full contract.

Channel Expiration

Bob can close the payment channel at any time, but if they fail to do so, Alice needs a way to recover her escrowed funds. An *expiration* time was set at the time of contract deployment. Once that time is reached, Alice can call `claimTimeout` to recover her funds. You can see the `claimTimeout` function in the full contract.

After this function is called, Bob can no longer receive any Ether, so it is important that Bob closes the channel before the expiration is reached.

The full contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Frozeable {
    bool private _frozen = false;

    modifier notFrozen() {
        require(!_frozen, "Inactive Contract.");
        _;
    }

    function freeze() internal {
        _frozen = true;
    }
}

contract SimplePaymentChannel is Frozeable {
    address payable public sender;    // The account sending payments.
    address payable public recipient; // The account receiving the payments.
    uint256 public expiration;        // Timeout in case the recipient never closes.

    constructor (address payable recipientAddress, uint256 duration)
        payable
    {
        sender = payable(msg.sender);
        recipient = recipientAddress;
        expiration = block.timestamp + duration;
    }

    /// the recipient can close the channel at any time by presenting a
    /// signed amount from the sender. the recipient will be sent that amount,
    /// and the remainder will go back to the sender
```

(continues on next page)

(continued from previous page)

```
    function close(uint256 amount, bytes memory signature)
        external
        notFrozen
    {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        freeze();
        (bool success, ) = recipient.call{value: amount}("");
        require(success);
        (success, ) = sender.call{value: address(this).balance}("");
        require(success);
    }

    /// the sender can extend the expiration at any time
    function extend(uint256 newExpiration)
        external
        notFrozen
    {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// if the timeout is reached without the recipient closing the channel,
    /// then the Ether is released back to the sender.
    function claimTimeout()
        external
        notFrozen
    {
        require(block.timestamp >= expiration);
        freeze();
        (bool success, ) = sender.call{value: address(this).balance}("");
        require(success);
    }

    function isValidSignature(uint256 amount, bytes memory signature)
        internal
        view
        returns (bool)
    {
        bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));
        // check that the signature is from the payment sender
        return recoverSigner(message, signature) == sender;
    }

    /// All functions below this are just taken from the chapter
    /// 'creating and verifying signatures' chapter.
    function splitSignature(bytes memory sig)
        internal
        pure
```

(continues on next page)

(continued from previous page)

```
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
            // second 32 bytes
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }
        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);
        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Note

The function `splitSignature` does not use all security checks. A real implementation should use a more rigorously tested library, such as openzeppelin's version of this code.

Verifying Payments

Unlike in the previous section, messages in a payment channel aren't redeemed right away. The recipient keeps track of the latest message and redeems it when it's time to close the payment channel. This means it's critical that the recipient perform their own verification of each message. Otherwise there is no guarantee that the recipient will be able to get paid in the end.

The recipient should verify each message using the following process:

1. Verify that the contract address in the message matches the payment channel.
2. Verify that the new total is the expected amount.
3. Verify that the new total does not exceed the amount of Ether escrowed.
4. Verify that the signature is valid and comes from the payment channel sender.

We'll use the ethereumjs-util library to write this verification. The final step can be done a number of ways, and we use JavaScript. The following code borrows the `constructPaymentMessage` function from the signing **JavaScript code**

above:

```
// this mimics the prefixing behavior of the eth_sign JSON-RPC method.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}
```

3.2.5 Modular Contracts

A modular approach to building your contracts helps you reduce the complexity and improve the readability which will help to identify bugs and vulnerabilities during development and code review. If you specify and control the behavior of each module in isolation, the interactions you have to consider are only those between the module specifications and not every other moving part of the contract. In the example below, the contract uses the `move` method of the `Balances` *library* to check that balances sent between addresses match what you expect. In this way, the `Balances` library provides an isolated component that properly tracks balances of accounts. It is easy to verify that the `Balances` library never produces negative balances or overflows and the sum of all balances is an invariant across the lifetime of the contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address to,
↪ uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping(address => uint256)) allowed;
```

(continues on next page)

(continued from previous page)

```
    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function transfer(address to, uint amount) external returns (bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;

    }

    function transferFrom(address from, address to, uint amount) external returns (bool␣
↪success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);
        emit Transfer(from, to, amount);
        return true;
    }

    function approve(address spender, uint tokens) external returns (bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }

    function balanceOf(address tokenOwner) external view returns (uint balance) {
        return balances[tokenOwner];
    }
}
```

3.3 Installing the Solidity Compiler

3.3.1 Versioning

Solidity versions follow Semantic Versioning. In addition, patch-level releases with major release 0 (i.e. 0.x.y) will not contain breaking changes. That means code that compiles with version 0.x.y can be expected to compile with 0.x.z where z > y.

In addition to releases, we provide **prereleases** and **nightly development builds** to make it easy for developers to try out upcoming features and provide early feedback. Note that such builds contain bleeding-edge code from the development branch and are not guaranteed to be of the same quality as full releases. Despite our best efforts, they might contain undocumented and/or broken changes that will not become a part of an actual release. They are not meant for production use.

When deploying contracts, you should use the latest released version of Solidity. This is because breaking changes, as well as new features and bug fixes are introduced regularly. We currently use a 0.x version number to indicate this fast pace of change.

3.3.2 Remix

We recommend Remix for small contracts and for quickly learning Solidity.

Access Remix online, you do not need to install anything. If you want to use it without connection to the Internet, download Remix Desktop from the releases page. Remix is also a convenient option for testing nightly builds without installing multiple Solidity versions.

Further options on this page detail installing command-line Solidity compiler software on your computer. Choose a command-line compiler if you are working on a larger contract or if you require more compilation options.

3.3.3 npm / Node.js

Use `npm` for a convenient and portable way to install `solcjs`, a Solidity compiler. The `solcjs` program has fewer features than the ways to access the compiler described further down this page. The *Using the Commandline Compiler* documentation assumes you are using the full-featured compiler, `solc`. The usage of `solcjs` is documented inside its own repository.

Note: The solc-js project is derived from the C++ `solc` by using Emscripten, which means that both use the same compiler source code. `solc-js` can be used in JavaScript projects directly (such as Remix). Please refer to the solc-js repository for instructions.

```
npm install --global solc
```

Note

The command-line executable is named `solcjs`.

The command-line options of `solcjs` are not compatible with `solc` and tools (such as `geth`) expecting the behavior of `solc` will not work with `solcjs`.

3.3.4 Docker

Docker images of Solidity builds are available using the solc image from the argotorg organization on ghcr.io. Use the `stable` tag for the latest released version, and `nightly` for potentially unstable changes in the `develop` branch.

The Docker image runs the compiler executable so that you can pass all compiler arguments to it. For example, the command below pulls the stable version of the `solc` image (if you do not have it already), and runs it in a new container, passing the `--help` argument.

```
docker run ghcr.io/argotorg/solc:stable --help
```

Note

Specific compiler versions are supported as the Docker image tag such as `ghcr.io/argotorg/solc:0.8.23`. We will be passing the `stable` tag here instead of specific version tag to ensure that users get the latest version by default and avoid the issue of an out-of-date version.

To use the Docker image to compile Solidity files on the host machine, mount a local folder for input and output, and specify the contract to compile. For example:

```
docker run \
    --volume "/tmp/some/local/path/:/sources/" \
    ghcr.io/argotorg/solc:stable \
```

(continues on next page)

(continued from previous page)

```
        /sources/Contract.sol \
        --abi \
        --bin \
        --output-dir /sources/output/
```

You can also use the standard JSON interface (which is recommended when using the compiler with tooling). When using this interface, it is not necessary to mount any directories as long as the JSON input is self-contained (i.e. it does not refer to any external files that would have to be *loaded by the import callback*).

```
docker run ghcr.io/argotorg/solc:stable --standard-json < input.json > output.json
```

3.3.5 Linux Packages

We provide *standalone binaries* of the compiler that should run on most distributions without any additional installation steps.

Ubuntu packages for versions up to 0.8.30 are available in the ethereum/ethereum PPA. However, we have discontinued this distribution method and future versions will not be added there.

Some Linux distributions provide their own packages. These packages are not directly maintained by us but usually kept up-to-date by the respective package maintainers.

Unofficial, community-maintained scripts for building and installing the compiler are also available for some distributions:

- Arch Linux / (AUR):
  - solidity (builds from source),
  - solidity-bin (uses our standalone binaries).
- Nix:
  - solc.nix (builds from source).

Note

Please be aware that these scripts are produced and maintained by users and not vetted in any way by the distro maintainers. Exercise caution when using them.

There is also a snap package, however, it is **currently unmaintained**. It is installable in all the supported Linux distros. To install the latest stable version of solc:

```
sudo snap install solc
```

If you want to help testing the latest development version of Solidity with the most recent changes, please use the following:

```
sudo snap install solc --edge
```

Note

The `solc` snap uses strict confinement. This is the most secure mode for snap packages but it comes with limitations, like accessing only the files in your `/home` and `/media` directories. For more information, go to Demystifying Snap Confinement.

3.3.6 macOS Packages

We distribute the Solidity compiler through Homebrew as a build-from-source version. Pre-built bottles are currently not supported.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

To install the most recent 0.4.x / 0.5.x version of Solidity you can also use `brew install solidity@4` and `brew install solidity@5`, respectively.

If you need a specific version of Solidity you can install a Homebrew formula directly from Github.

View solidity.rb commits on GitHub.

Copy the commit hash of the version you want and check it out on your machine.

```
git clone https://github.com/ethereum/homebrew-ethereum.git
cd homebrew-ethereum
git checkout <your-hash-goes-here>
```

Install it using `brew`:

```
brew unlink solidity
# eg. Install 0.4.8
brew install solidity.rb
```

3.3.7 Static Binaries

We maintain a repository containing static builds of past and current compiler versions for all supported platforms at solc-bin. This is also the location where you can find the nightly builds.

The repository is not only a quick and easy way for end users to get binaries ready to be used out-of-the-box but it is also meant to be friendly to third-party tools:

- The content is mirrored to https://binaries.soliditylang.org where it can be easily downloaded over HTTPS without any authentication, rate limiting or the need to use git.
- Content is served with correct `Content-Type` headers and lenient CORS configuration so that it can be directly loaded by tools running in the browser.
- Binaries do not require installation or unpacking (exception for older Windows builds bundled with necessary DLLs).
- We strive for a high level of backward-compatibility. Files, once added, are not removed or moved without providing a symlink/redirect at the old location. They are also never modified in place and should always match the original checksum. The only exception would be broken or unusable files with the potential to cause more harm than good if left as is.
- Files are served over both HTTP and HTTPS. As long as you obtain the file list in a secure way (via git, HTTPS, IPFS or just have it cached locally) and verify hashes of the binaries after downloading them, you do not have to use HTTPS for the binaries themselves.

The same binaries are in most cases available on the Solidity release page on GitHub. The difference is that we do not generally update old releases on the GitHub release page. This means that we do not rename them if the naming convention changes and we do not add builds for platforms that were not supported at the time of release. This only happens in `solc-bin`.

The `solc-bin` repository contains several top-level directories, each representing a single platform. Each one includes a `list.json` file listing the available binaries. For example in `emscripten-wasm32/list.json` you will find the following information about version 0.7.4:

```
{
  "path": "solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js",
  "version": "0.7.4",
  "build": "commit.3f05b770",
  "longVersion": "0.7.4+commit.3f05b770",
  "keccak256": "0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3",
  "sha256": "0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2",
  "urls": [
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS"
  ]
}
```

This means that:

- You can find the binary in the same directory under the name solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. Note that the file might be a symlink, and you will need to resolve it yourself if you are not using git to download it or your file system does not support symlinks.
- The binary is also mirrored at https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. In this case git is not necessary and symlinks are resolved transparently, either by serving a copy of the file or returning a HTTP redirect.
- The file is also available on IPFS at QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS. Please, be aware that the order of items in the `urls` array is not predetermined or guaranteed and users should not rely on it.
- You can verify the integrity of the binary by comparing its keccak256 hash to `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3`. The hash can be computed on the command-line using `keccak256sum` utility provided by sha3sum or keccak256() function from ethereumjs-util in JavaScript.
- You can also verify the integrity of the binary by comparing its sha256 hash to `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2`.

Warning

Due to the strong backwards compatibility requirement the repository contains some legacy elements but you should avoid using them when writing new tools:

- Use `emscripten-wasm32/` (with a fallback to `emscripten-asmjs/`) instead of `bin/` if you want the best performance. Until version 0.6.1 we only provided asm.js binaries. Starting with 0.6.2 we switched to WebAssembly builds with much better performance. We have rebuilt the older versions for wasm but the original asm.js files remain in `bin/`. The new ones had to be placed in a separate directory to avoid name clashes.
- Use `emscripten-asmjs/` and `emscripten-wasm32/` instead of `bin/` and `wasm/` directories if you want to be sure whether you are downloading a wasm or an asm.js binary.
- Use `list.json` instead of `list.js` and `list.txt`. The JSON list format contains all the information from the old ones and more.

Warning

- The solc-bin.ethereum.org domain is no longer supported. Going forward, we recommend any tools which are still using it as the source of Solidity binaries to switch to binaries.soliditylang.org.

Warning

The binaries are also available at https://argotorg.github.io/solc-bin/ but this page stopped being updated just after the release of version 0.7.2, will not receive any new releases or nightly builds for any platform and does not serve the new directory structure, including non-emscripten builds.

If you are using it, please switch to https://binaries.soliditylang.org, which is a drop-in replacement. This allows us to make changes to the underlying hosting in a transparent way and minimize disruption. Unlike the `argotorg.github.io` domain, which we do not have any control over, `binaries.soliditylang.org` is guaranteed to work and maintain the same URL structure in the long-term.

3.3.8 Building from Source

Prerequisites - All Operating Systems

The following are dependencies for all builds of Solidity:

Software	Notes
CMake (version 3.21.3+ on Windows, 3.13+ otherwise)	Cross-platform build file generator.
Boost (version 1.77+ on Windows, 1.83+ otherwise)	C++ libraries.
Git	Command-line tool for retrieving source code.
z3 (version 4.8.16+, Optional)	For use with SMT checker.

Note

Solidity versions prior to 0.5.10 can fail to correctly link against Boost versions 1.70+. A possible workaround is to temporarily rename `<Boost install path>/lib/cmake/Boost-1.70.0` prior to running the cmake command to configure Solidity.

Starting from 0.5.10 linking against Boost 1.70+ should work without manual intervention.

Note

The default build configuration requires a specific Z3 version (the latest one at the time the code was last updated). Changes introduced between Z3 releases often result in slightly different (but still valid) results being returned. Our SMT tests do not account for these differences and will likely fail with a different version than the one they were written for. This does not mean that a build using a different version is faulty. If you pass `-DSTRICT_Z3_VERSION=OFF` option to CMake, you can build with any version that satisfies the requirement given in the table above. If you do this, however, please remember to pass the `--no-smt` option to `scripts/tests.sh` to skip the SMT tests.

Note

By default the build is performed in *pedantic mode*, which enables extra warnings and tells the compiler to treat all warnings as errors. This forces developers to fix warnings as they arise, so they do not accumulate "to be fixed later".

If you are only interested in creating a release build and do not intend to modify the source code to deal with such warnings, you can pass `-DPEDANTIC=OFF` option to CMake to disable this mode. Doing this is not recommended for general use but may be necessary when using a toolchain we are not testing with or trying to build an older version with newer tools. If you encounter such warnings, please consider reporting them.

Minimum Compiler Versions

The following C++ compilers and their minimum versions can build the Solidity codebase:

- GCC, version 13.3+
- Clang, version 18.1.3+
- MSVC, version 2019+

Prerequisites - macOS

For macOS builds, ensure that you have the latest version of Xcode installed. This contains the Clang C++ compiler, the Xcode IDE and other Apple development tools that are required for building C++ applications on OS X. If you are installing Xcode for the first time, or have just installed a new version then you will need to agree to the license before you can do command-line builds:

```
sudo xcodebuild -license accept
```

Our OS X build script uses the Homebrew package manager for installing external dependencies. Here's how to uninstall Homebrew, if you ever want to start again from scratch.

Prerequisites - Windows

You need to install the following dependencies for Windows builds of Solidity:

Software	Notes
Visual Studio 2019 Build Tools	C++ compiler
Visual Studio 2019 (Optional)	C++ compiler and dev environment.
Boost (version 1.77+)	C++ libraries.

If you already have one IDE and only need the compiler and libraries, you could install Visual Studio 2019 Build Tools.

Visual Studio 2019 provides both IDE and necessary compiler and libraries. So if you have not got an IDE and prefer to develop Solidity, Visual Studio 2019 may be a choice for you to get everything setup easily.

Here is the list of components that should be installed in Visual Studio 2019 Build Tools or Visual Studio 2019:

- Visual Studio C++ core features
- VC++ 2019 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- C++/CLI support

We have a helper script which you can use to install all required external dependencies:

```
scripts\install_deps.ps1
```

This will install `boost` and `cmake` to the `deps` subdirectory.

Clone the Repository

To clone the source code, execute the following command:

```
git clone --recursive https://github.com/argotorg/solidity.git
cd solidity
```

If you want to help develop Solidity, you should fork Solidity and add your personal fork as a second remote:

```
git remote add personal git@github.com:[username]/solidity.git
```

Note

This method will result in a pre-release build leading to e.g. a flag being set in each bytecode produced by such a compiler. If you want to re-build a released Solidity compiler, then please use the source tarball on the GitHub release page:

https://github.com/argotorg/solidity/releases/download/v0.X.Y/solidity_0.X.Y.tar.gz

(not the "Source code" provided by GitHub).

Command-Line Build

Be sure to install External Dependencies (see above) before build.

Solidity project uses CMake to configure the build. You might want to install ccache to speed up repeated builds. CMake will pick it up automatically. Building Solidity is quite similar on Linux, macOS and other Unices:

```
mkdir build
cd build
cmake .. && make
```

or even easier on Linux and macOS, you can run:

```
#note: this will install binaries solc and soltest at usr/local/bin
./scripts/build.sh
```

Warning

BSD builds should work, but are untested by the Solidity team.

And for Windows:

```
mkdir build
cd build
cmake -G "Visual Studio 16 2019" ..
```

In case you want to use the version of boost installed by `scripts\install_deps.ps1`, you will additionally need to pass `-DBoost_ROOT="deps/boost" -DBoost_INCLUDE_DIR="deps/boost/include"` and `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` as arguments to the call to `cmake`.

This should result in the creation of **solidity.sln** in that build directory. Double-clicking on that file should result in Visual Studio firing up. We suggest building **Release** configuration, but all others work.

Alternatively, you can build for Windows on the command-line, like so:

```
cmake --build . --config Release
```

3.3.9 CMake Options

If you are interested what CMake options are available run `cmake .. -LH`.

SMT Solvers

Solidity can optionally use SMT solvers, namely `z3`, `cvc5` and `Eldarica`, but their presence is checked only at runtime, they are not needed for the build to succeed.

> **Note**
>
> The emscripten builds require Z3 and will statically link against it instead.

3.3.10 The Version String in Detail

The Solidity version string contains four parts:

- the version number
- pre-release tag, usually set to `develop.YYYY.MM.DD`, `pre.N` or `nightly.YYYY.MM.DD`
- commit in the format of `commit.GITHASH`
- platform, which has an arbitrary number of items, containing details about the platform and compiler

If there are local modifications, the commit will be postfixed with `.mod`.

These parts are combined as required by SemVer, where the Solidity pre-release tag equals to the SemVer pre-release and the Solidity commit and platform combined make up the SemVer build metadata.

Examples:

- release: `0.4.8+commit.60cc1668.Emscripten.clang`
- pre-release: `0.4.9-pre.3+commit.fb60450bc.Emscripten.clang`
- nightly build: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

3.3.11 Important Information About Versioning

After a release is made, the patch version level is bumped, because we assume that only patch level changes follow. When changes are merged, the version should be bumped according to SemVer and the severity of the change. Finally, a release is always made with the version of the current build, but without the `prerelease` specifier.

Example:

1. The 0.4.0 release is made.
2. Nightly builds and preerelases have a version of 0.4.1 from now on.
3. Non-breaking changes are introduced –> no change in version.
4. A breaking change is introduced –> version is bumped to 0.5.0.
5. The 0.5.0 release is made.

This behavior works well with the *version pragma*.

3.4 Layout of a Solidity Source File

Source files can contain an arbitrary number of *contract definitions*, *import* , *pragma* and *using for* directives and *struct*, *enum*, *function*, *error* and *constant variable* definitions.

3.4.1 SPDX License Identifier

Trust in smart contracts can be better established if their source code is available. Since making source code available always touches on legal problems with regards to copyright, the Solidity compiler encourages the use of machine-readable SPDX license identifiers. Every source file should start with a comment indicating its license:

```
// SPDX-License-Identifier:  MIT
```

The compiler does not validate that the license is part of the list allowed by SPDX, but it does include the supplied string in the *bytecode metadata*.

If you do not want to specify a license or if the source code is not open-source, please use the special value `UNLICENSED`. Note that `UNLICENSED` (no usage allowed, not present in SPDX license list) is different from `UNLICENSE` (grants all rights to everyone). Solidity follows the npm recommendation.

Supplying this comment of course does not free you from other obligations related to licensing like having to mention a specific license header in each source file or the original copyright holder.

The comment is recognized by the compiler anywhere in the file at the file level, but it is recommended to put it at the top of the file.

More information about how to use SPDX license identifiers can be found at the SPDX website.

3.4.2 Pragmas

The `pragma` keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in your whole project. If you *import* another file, the pragma from that file does *not* automatically apply to the importing file.

Version Pragma

Source files can (and should) be annotated with a version pragma to reject compilation with future compiler versions that might introduce incompatible changes. We try to keep these to an absolute minimum and introduce them in a way that changes in semantics also require changes in the syntax, but this is not always possible. Because of this, it is always a good idea to read through the changelog at least for releases that contain breaking changes. These releases always have versions of the form `0.x.0` or `x.0.0`.

The version pragma is used as follows: `pragma solidity ^0.5.2;`

A source file with the line above does not compile with a compiler earlier than version 0.5.2, and it also does not work on a compiler starting from version 0.6.0 (this second condition is added by using `^`). Because there will be no breaking changes until version `0.6.0`, you can be sure that your code compiles the way you intended. The exact version of the compiler is not fixed, so that bugfix releases are still possible.

It is possible to specify more complex rules for the compiler version, these follow the same syntax used by npm.

> **Note**
>
> Using the version pragma *does not* change the version of the compiler. It also *does not* enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.

ABI Coder Pragma

By using `pragma abicoder v1` or `pragma abicoder v2` you can select between the two implementations of the ABI encoder and decoder.

The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. Apart from supporting more types, it involves more extensive validation and safety checks, which may result in higher gas costs, but also heightened security. It is considered non-experimental as of Solidity 0.6.0 and it is enabled by default starting with Solidity 0.8.0. The old ABI coder can still be selected using `pragma abicoder v1;`.

Warning

The ABI coder v1 is deprecated and scheduled for removal. Use ABI coder v2 instead.

The set of types supported by the new encoder is a strict superset of the ones supported by the old one. Contracts that use it can interact with ones that do not without limitations. The reverse is possible only as long as the non-`abicoder v2` contract does not try to make calls that would require decoding types only supported by the new encoder. The compiler can detect this and will issue an error. Simply enabling `abicoder v2` for your contract is enough to make the error go away.

Note

This pragma applies to all the code defined in the file where it is activated, regardless of where that code ends up eventually. This means that a contract whose source file is selected to compile with ABI coder v1 can still contain code that uses the new encoder by inheriting it from another contract. This is allowed if the new types are only used internally and not in external function signatures.

Note

Up to Solidity 0.7.4, it was possible to select the ABI coder v2 by using `pragma experimental ABIEncoderV2`, but it was not possible to explicitly select coder v1 because it was the default.

Experimental Pragma

The second pragma is the experimental pragma. It can be used to enable features of the compiler or language that are not yet enabled by default. The following experimental pragmas are currently supported:

ABIEncoderV2

Because the ABI coder v2 is not considered experimental anymore, it can be selected via `pragma abicoder v2` (please see above) since Solidity 0.7.4.

SMTChecker

If you use `pragma experimental SMTChecker;`, then you get additional *safety warnings* which are obtained by querying an SMT solver. The component does not yet support all features of the Solidity language and likely outputs many warnings. In case it reports unsupported features, the analysis may not be fully sound.

Note

The `SMTChecker` pragma is deprecated and will be removed. To enable SMTChecker, simply select *select an engine* when invoking the compiler.

3.4.3 Importing other Source Files

Syntax and Semantics

Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on). However, Solidity does not support the concept of a default export.

At a global level, you can use import statements of the following form:

```
import "filename";
```

The `filename` part is called an *import path*. This statement imports all global symbols from "filename" (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity). This form is not recommended for use, because it unpredictably pollutes the namespace. If you add new top-level items inside "filename", they automatically appear in all files that import like this from "filename". It is better to import specific symbols explicitly.

The following example creates a new global symbol `symbolName` whose members are all the global symbols from `"filename"`:

```
import * as symbolName from "filename";
```

which results in all global symbols being available in the format `symbolName.symbol`.

A variant of this syntax that is not part of ES6, but possibly useful is:

```
import "filename" as symbolName;
```

which is equivalent to `import * as symbolName from "filename";`.

If there is a naming collision, you can rename symbols while importing. For example, the code below creates new global symbols `alias` and `symbol2` which reference `symbol1` and `symbol2` from inside `"filename"`, respectively.

```
import {symbol1 as alias, symbol2} from "filename";
```

Import Paths

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. For this reason import paths do not refer directly to files in the host filesystem. Instead the compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. The import path specified in an import statement is translated into a source unit name and used to find the corresponding source unit in this database.

Using the *Standard JSON* API it is possible to directly provide the names and content of all the source files as a part of the compiler input. In this case source unit names are truly arbitrary. If, however, you want the compiler to automatically find and load source code into the VFS, your source unit names need to be structured in a way that makes it possible for an *import callback* to locate them. When using the command-line compiler the default import callback supports only loading source code from the host filesystem, which means that your source unit names must be paths. Some environments provide custom callbacks that are more versatile. For example the Remix IDE provides one that lets you import files from HTTP, IPFS and Swarm URLs or refer directly to packages in NPM registry.

For a complete description of the virtual filesystem and the path resolution logic used by the compiler see *Path Resolution*.

3.4.4 Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Note

A single-line comment is terminated by any unicode line terminator (LF, VF, FF, CR, NEL, LS or PS) in UTF-8 encoding. The terminator is still part of the source code after the comment, so if it is not an ASCII symbol (these are NEL, LS and PS), it will lead to a parser error.

Additionally, there is another type of comment called a NatSpec comment, which is detailed in the *style guide*. They are written with a triple slash (`///`) or a double asterisk block (`/** ... */`) and they should be used directly above function declarations or statements.

3.5 Structure of a Contract

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of *State Variables*, *Functions*, *Function Modifiers*, *Events*, *Errors*, *Struct Types* and *Enum Types*. Furthermore, contracts can inherit from other contracts.

There are also special kinds of contracts called *libraries* and *interfaces*.

The section about *contracts* contains more details than this section, which serves to provide a quick overview.

3.5.1 State Variables

State variables are variables whose values are either permanently stored in contract storage or, alternatively, temporarily stored in transient storage which is cleaned at the end of each transaction. See *data locations* for more details.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

See the *Types* section for valid state variable types and *Visibility and Getters* for possible choices for visibility.

3.5.2 Functions

Functions are the executable units of code. Functions are usually defined inside a contract, but they can also be defined outside of contracts.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;
```

(continues on next page)

(continued from previous page)

```
contract SimpleAuction {
    function bid() public payable { // Function
        // ...
    }
}

// Helper function defined outside of a contract
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Function Calls can happen internally or externally and have different levels of *visibility* towards other contracts. *Functions* accept *parameters and return variables* to pass parameters and values between them.

3.5.3 Function Modifiers

Function modifiers can be used to amend the semantics of functions in a declarative way (see *Function Modifiers* in the contracts section).

Overloading, that is, having the same modifier name with different parameters, is not possible.

Like functions, modifiers can be *overridden*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public view onlySeller { // Modifier usage
        // ...
    }
}
```

3.5.4 Events

Events are convenience interfaces with the EVM logging facilities.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.22;

event HighestBidIncreased(address bidder, uint amount); // Event

contract SimpleAuction {
```

(continues on next page)

(continued from previous page)

```
    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // Triggering event
    }
}
```

See *Events* in contracts section for information on how events are declared and can be used from within a dapp.

3.5.5 Errors

Errors allow you to define descriptive names and data for failure situations. Errors can be used in *revert statements*. In comparison to string descriptions, errors are much cheaper and allow you to encode additional data. You can use NatSpec to describe the error to the user.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Not enough funds for transfer. Requested `requested`,
/// but only `available` available.
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
        // ...
    }
}
```

See *Custom Errors* in the contracts section for more information.

3.5.6 Struct Types

Structs are custom defined types that can group several variables (see *Structs* in types section).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

3.5.7 Enum Types

Enums can be used to create custom types with a finite set of 'constant values' (see *Enums* in types section).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

3.6 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary types which can be combined to form complex types.

In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see *Order of Precedence of Operators*.

The concept of "undefined" or "null" values does not exist in Solidity, but newly declared variables always have a *default value* dependent on its type. To handle any unexpected values, you should use the *revert function* to revert the whole transaction, or return a tuple with a second `bool` value denoting success.

3.6.1 Value Types

The following are called value types because their variables will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Unlike *reference types*, value type declarations do not specify a data location since they are small enough to be stored on the stack. The only exception is *state variables*. Those are by default located in storage, but can also be marked as *transient*, *constant or immutable*.

Booleans

`bool`: The possible values are constants `true` and `false`.

Operators:

- ! (logical negation)
- && (logical conjunction, "and")
- || (logical disjunction, "or")
- == (equality)
- != (inequality)

The operators `||` and && apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`int` / `uint`: Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of `8` (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)

- Shift operators: << (left shift), >> (right shift)
- Arithmetic operators: +, -, unary - (only for signed integers), *, /, % (modulo), ** (exponentiation)

For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

Warning

Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is `0` up to `2**32 - 1`. There are two modes in which arithmetic is performed on these types: The "wrapping" or "unchecked" mode and the "checked" mode. By default, arithmetic is always "checked", meaning that if an operation's result falls outside the value range of the type, the call is reverted through a *failing assertion*. You can switch to "unchecked" mode using `unchecked { ... }`. More details can be found in the section about *unchecked*.

Comparisons

The value of a comparison is the one obtained by comparing the integer value.

Bit operations

Bit operations are performed on the two's complement representation of the number. This means that, for example `~int256(0) == int256(-1)`.

Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by a signed type will produce a compilation error.

Shifts can be "simulated" using multiplication by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

- `x << y` is equivalent to the mathematical expression `x * 2**y`.
- `x >> y` is equivalent to the mathematical expression `x / 2**y`, rounded towards negative infinity.

Warning

Before version `0.5.0` a right shift `x >> y` for negative `x` was equivalent to the mathematical expression `x / 2**y` rounded towards zero, i.e., right shifts used rounding up (towards zero) instead of rounding down (towards negative infinity).

Note

Overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated.

Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics, with two different modes in regard to over- and underflow:

By default, all arithmetic is checked for under- or overflow, but this can be disabled using the *unchecked block*, resulting in wrapping arithmetic. More details can be found in that section.

The expression `-x` is equivalent to `(T(0) - x)` where `T` is the type of `x`. It can only be applied to signed types. The value of `-x` can be positive if `x` is negative. There is another caveat also resulting from two's complement representation:

If you have `int x = type(int).min;`, then `-x` does not fit the positive range. This means that `unchecked { assert(-x == x); }` works, and the expression `-x` when used in checked mode will result in a failing assertion.

Division

Since the type of the result of an operation is always the type of one of the operands, division on integers always results in an integer. In Solidity, division rounds towards zero. This means that `int256(-5) / int256(2) == int256(-2)`.

Note that in contrast, division on *literals* results in fractional values of arbitrary precision.

Note

Division by zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

Note

The expression `type(int).min / (-1)` is the only case where division causes an overflow. In checked arithmetic mode, this will cause a failing assertion, while in wrapping mode, the value will be `type(int).min`.

Modulo

The modulo operation `a % n` yields the remainder `r` after the division of the operand `a` by the operand `n`, where `q = int(a / n)` and `r = a - (n * q)`. This means that modulo results in the same sign as its left operand (or zero) and `a % n == -(-a % n)` holds for negative `a`:

- `int256(5) % int256(2) == int256(1)`
- `int256(5) % int256(-2) == int256(1)`
- `int256(-5) % int256(2) == int256(-1)`
- `int256(-5) % int256(-2) == int256(-1)`

Note

Modulo with zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential assertion failures or wrapping behavior.

Note

In checked mode, exponentiation only uses the comparatively cheap `exp` opcode for small bases. For the cases of `x**3`, the expression `x*x*x` might be cheaper. In any case, gas cost tests and the use of the optimizer are advisable.

Note

Note that `0**0` is defined by the EVM as 1.

Fixed Point Numbers

Warning

Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from.

`fixed` / `ufixed`: Signed and unsigned fixed point number of various sizes. Keywords `ufixedMxN` and `fixedMxN`, where `M` represents the number of bits taken by the type and `N` represents how many decimal points are available. `M` must be divisible by 8 and goes from 8 to 256 bits. `N` must be between 0 and 80, inclusive. `ufixed` and `fixed` are aliases for `ufixed128x18` and `fixed128x18`, respectively.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Arithmetic operators: +, -, unary -, *, /, % (modulo)

Note

The main difference between floating point (`float` and `double` in many languages, more precisely IEEE 754 numbers) and fixed point numbers is that the number of bits used for the integer and the fractional part (the part after the decimal dot) is flexible in the former, while it is strictly defined in the latter. Generally, in floating point almost the entire space is used to represent the number, while only a small number of bits define where the decimal point is.

Address

The address type comes in two largely identical flavors:

- `address`: Holds a 20 byte value (size of an Ethereum address).
- `address payable`: Same as `address`, but with the additional members `transfer` and `send`.

The idea behind this distinction is that `address payable` is an address you can send Ether to, while you are not supposed to send Ether to a plain `address`, for example because it might be a smart contract that was not built to accept Ether.

Type conversions:

Implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(<address>)`.

Explicit conversions to and from `address` are allowed for `uint160`, integer literals, `bytes20` and contract types.

Only expressions of type `address` and contract type can be converted to the type `address payable` via the explicit conversion `payable(...)`. For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a *receive* or a payable fallback function. Note that `payable(0)` is valid and is an exception to this rule.

Note

If you need a variable of type `address` and plan to send Ether to it, then declare its type as `address payable` to make this requirement visible. Also, try to make this distinction or conversion as early as possible.

The distinction between `address` and `address payable` was introduced in version 0.5.0. Also starting from that version, contracts are not implicitly convertible to the `address` type, but can still be explicitly converted to `address` or to `address payable`, if they have a receive or payable fallback function.

Operators:

- <=, <, ==, !=, >= and >

Warning

If you convert a type that uses a larger byte size to an `address`, for example `bytes32`, then the `address` is truncated. To reduce conversion ambiguity, starting with version 0.4.24, the compiler will force you to make the truncation explicit in the conversion. Take for example the 32-byte value `0x1111222233334444555566667777888899999AAAABBBBCCCCDDDDEEEEFFFFCCCC`.

You can use `address(bytes20(b))`, which results in `0x1111222233334444555566667777888899999aAaa`, or you can use `address(uint160(uint256(b)))`, which results in `0x777788889999AaAAbBbbCcccddDdeeeEfFFfCcCc`.

Note

Mixed-case hexadecimal numbers conforming to EIP-55 are automatically treated as literals of the `address` type. See *Address Literals*.

Members of Addresses

For a quick reference of all members of address, see *Members of Address Types*.

- `balance` and `transfer`

  It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to a payable address using the `transfer` function:

  ```
  address payable x = payable(0x123);
  address myAddress = address(this);
  if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
  ```

  The `transfer` function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The `transfer` function reverts on failure.

  Note

  If `x` is a contract address, its code (more specifically: its *Receive Ether Function*, if present, or otherwise its *Fallback Function*, if present) will be executed together with the `transfer` call (this is a feature of the EVM and cannot be prevented). If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

> **Warning**
>
> `transfer` is deprecated and scheduled for removal. Simple ether transfers can still be performed using the *call function* with with an optionally provided maximum amount of gas and empty payload, i.e., `call{value: <amount>}("")`. By default this forwards all the remaining gas, subject to additional limits imposed by some EVM versions (such as the 63/64th rule introduced by `tangerineWhistle`). As with any external call, the `gas` call option can be used to set a lower limit.
>
> While it is possible to recreate the functionality by explicitly setting the limit to the value of the stipend (2300 gas), this value no longer holds its original meaning due to changing opcode costs. It is recommended to use different means to protect against reentrancy.

- `send`

  `send` is the low-level counterpart of `transfer`. If the execution fails, the current contract will not stop with an exception, but `send` will return `false`.

  > **Warning**
  >
  > There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: use a pattern where the recipient withdraws the Ether.

  > **Warning**
  >
  > `send` is deprecated and scheduled for removal. Simple ether transfers can still be performed using the *call function* with with an optionally provided maximum amount of gas and empty payload, i.e., `call{value: <amount>}("")`. By default this forwards all the remaining gas, subject to additional limits imposed by some EVM versions (such as the 63/64th rule introduced by `tangerineWhistle`). As with any external call, the `gas` call option can be used to set a lower limit.
  >
  > While it is possible to recreate the functionality by explicitly setting the limit to the value of the stipend (2300 gas), this value no longer holds its original meaning due to changing opcode costs. It is recommended to use different means to protect against reentrancy.

- `call`, `delegatecall` and `staticcall`

  In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single `bytes memory` parameter and return the success condition (as a `bool`) and the returned data (`bytes memory`). The functions `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` and `abi.encodeWithSignature` can be used to encode structured data.

  Example:

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName
↪");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

Warning

All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns. The regular way to interact with other contracts is to call a function on a contract object (`x.f()`).

Note

Previous versions of Solidity allowed these functions to receive arbitrary arguments and would also handle a first argument of type `bytes4` differently. These edge cases were removed in version 0.5.0.

It is possible to adjust the supplied gas with the `gas` modifier:

```
address(nameReg).call{gas: 1000000}(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

Similarly, the supplied Ether value can be controlled too:

```
address(nameReg).call{value: 1 ether}(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

Lastly, these modifiers can be combined. Their order does not matter:

```
address(nameReg).call{gas: 1000000, value: 1 ether}(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

In a similar way, the function `delegatecall` can be used: the difference is that only the code of the given address is used, all other aspects (storage, balance, . . .) are taken from the current contract. The purpose of `delegatecall` is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for delegatecall to be used.

Note

Prior to homestead, only a limited variant called `callcode` was available that did not provide access to the original `msg.sender` and `msg.value` values. This function was removed in version 0.5.0.

Since byzantium `staticcall` can be used as well. This is basically the same as `call`, but will revert if the called function modifies the state in any way.

All three functions `call`, `delegatecall` and `staticcall` are very low-level functions and should only be used as a *last resort* as they break the type-safety of Solidity.

The `gas` option is available on all three methods, while the `value` option is only available on `call`.

Note

It is best to avoid relying on hardcoded gas values in your smart contract code, regardless of

whether state is read from or written to, as this can have many pitfalls. Also, access to gas might change in the future.

- `code` and `codehash`

  You can query the deployed code for any smart contract. Use `.code` to get the EVM bytecode as a `bytes memory`, which might be empty. Use `.codehash` to get the Keccak-256 hash of that code (as a `bytes32`). Note that `addr.codehash` is cheaper than using `keccak256(addr.code)`.

Warning

The output of `addr.codehash` may be `0` if the account associated with `addr` is empty or non-existent (i.e., it has no code, zero balance, and zero nonce as defined by EIP-161). If the account has no code but a non-zero balance or nonce, then `addr.codehash` will output the Keccak-256 hash of empty data (i.e., `keccak256("")` which is equal to `c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`), as defined by EIP-1052.

Note

All contracts can be converted to `address` type, so it is possible to query the balance of the current contract using `address(this).balance`.

Contract Types

Every *contract* defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the `address` type.

Explicit conversion to and from the `address payable` type is only possible if the contract type has a receive or payable fallback function. The conversion is still performed using `address(x)`. If the contract type does not have a receive or payable fallback function, the conversion to `address payable` can be done using `payable(address(x))`. You can find more information in the section about the *address type*.

Note

Before version 0.5.0, contracts directly derived from the address type and there was no distinction between `address` and `address payable`.

If you declare a local variable of contract type (`MyContract c`), you can call functions on that contract. Take care to assign it from somewhere that is the same contract type.

You can also instantiate contracts (which means they are newly created). You can find more details in the *'Contracts via new'* section.

The data representation of a contract is identical to that of the `address` type and this type is also used in the *ABI*.

Contracts do not support any operators.

The members of contract types are the external functions of the contract including any state variables marked as `public`.

For a contract `C` you can use `type(C)` to access *type information* about the contract.

Fixed-size byte arrays

The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from one to up to 32.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), ~ (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

The shifting operator works with unsigned integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a signed type will produce a compilation error.

Members:

- `.length` yields the fixed length of the byte array (read-only).

Note

The type `bytes1[]` is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the `bytes` type instead.

Note

Prior to version 0.8.0, `byte` used to be an alias for `bytes1`.

Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of `address` type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

Note

The mixed-case address checksum format is defined in EIP-55.

Rational and Integer Literals

Integer literals are formed from a sequence of digits in the range 0-9. They are interpreted as decimals. For example, `69` means sixty nine. Octal literals do not exist in Solidity and leading zeros are invalid.

Decimal fractional literals are formed by a `.` with at least one number after the decimal point. Examples include `.1` and `1.3` (but not `1.`).

Scientific notation in the form of `2e10` is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal `MeE` is equivalent to `M * 10**E`. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`.

Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal `123_000`, hexadecimal `0x2eff_abde`, scientific decimal notation `1_2e345_678` are all valid. Underscores are only allowed between two digits and only one consecutive underscore is allowed. There is no additional semantic meaning added to a number literal containing underscores, the underscores are ignored.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with anything other than a number literal expression (like boolean literals) or by explicit conversion). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, `(2**800 + 1) - 2**800` results in the constant `1` (of type `uint8`) although intermediate results would not even fit the machine word size. Furthermore, `.5 * 8` results in the integer `4` (although non-integers were used in between).

Warning

While most operators produce a literal expression when applied to literals, there are certain operators that do not follow this pattern:

- Ternary operator (`... ? ... : ...`),
- Array subscript (`<array>[<index>]`).

You might expect expressions like `255 + (true ? 1 : 0)` or `255 + [1, 2, 3][0]` to be equivalent to using the literal 256 directly, but in fact they are computed within the type `uint8` and can overflow.

Any operator that can be applied to integers can also be applied to number literal expressions as long as the operands are integers. If any of the two is fractional, bit operations are disallowed and exponentiation is disallowed if the exponent is fractional (because that might result in a non-rational number).

Shifts and exponentiation with literal numbers as left (or base) operand and integer types as the right (exponent) operand are always performed in the `uint256` (for non-negative literals) or `int256` (for a negative literals) type, regardless of the type of the right (exponent) operand.

Warning

Division on integer literals used to truncate in Solidity prior to version 0.4.0, but it now converts into a rational number, i.e. `5 / 2` is not equal to `2`, but to `2.5`.

Note

Solidity has a number literal type for each rational number. Integer literals and rational number literals belong to number literal types. Moreover, all number literal expressions (i.e. the expressions that contain only number literals and operators) belong to number literal types. So the number literal expressions `1 + 2` and `2 + 1` both belong to the same number literal type for the rational number three.

Note

Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Disregarding types, the value of the expression assigned to `b` below evaluates to an integer. Because `a` is of type `uint128`, the expression `2.5 + a` has to have a proper type, though. Since there is no common type for the type of `2.5` and `uint128`, the Solidity compiler does not accept this code.

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

String Literals and Types

String literals are written with either double or single-quotes (`"foo"` or `'bar'`), and they can also be split into multiple consecutive parts (`"foo" "bar"` is equivalent to `"foobar"`) which can be helpful when dealing with long strings. They do not imply trailing zeroes as in C; `"foo"` represents three bytes, not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, ..., `bytes32`, if they fit, to `bytes` and to `string`.

For example, with `bytes32 samevar = "stringliteral"` the string literal is interpreted in its raw byte form when assigned to a `bytes32` type.

String literals can only contain printable ASCII characters, which means the characters between and including 0x20 .. 0x7E.

Additionally, string literals also support the following escape characters:

- `\<newline>` (escapes an actual newline)
- `\\` (backslash)
- `\'` (single quote)
- `\"` (double quote)
- `\n` (newline)
- `\r` (carriage return)
- `\t` (tab)
- `\xNN` (hex escape, see below)
- `\uNNNN` (unicode escape, see below)

`\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

> **Note**
>
> Until version 0.8.0 there were three additional escape sequences: `\b`, `\f` and `\v`. They are commonly available in other languages but rarely needed in practice. If you do need them, they can still be inserted via hexadecimal escapes, i.e. `\x08`, `\x0c` and `\x0b`, respectively, just as any other ASCII character.

The string in the following example has a length of ten bytes. It starts with a newline byte, followed by a double quote, a single quote a backslash character and then (without separator) the character sequence `abcdef`.

```
"\n\"\'\\abc\
def"
```

Any Unicode line terminator which is not a newline (i.e. LF, VF, FF, CR, NEL, LS, PS) is considered to terminate the string literal. Newline only terminates the string literal if it is not preceded by a `\`.

Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword `unicode` – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

```
string memory a = unicode"Hello ";
```

Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`, `hex'0011_22_FF'`). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: `hex"00112233" hex"44556677"` is equivalent to `hex"0011223344556677"`

Hexadecimal literals in some ways behave like *string literals* but are not implicitly convertible to the `string` type.

Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a *Panic error* otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

The data representation is the same as for enums in C: The options are represented by subsequent unsigned integer values starting from `0`.

Using `type(NameOfEnum).min` and `type(NameOfEnum).max` you can get the smallest and respectively largest value of the given enum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }

    function getLargestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).max;
    }

    function getSmallestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).min;
    }
}
```

Note

Enums can also be declared on the file level, outside of contract or library definitions.

User-defined Value Types

A user-defined value type allows creating a zero cost abstraction over an elementary value type. This is similar to an alias, but with stricter type requirements.

A user-defined value type is defined using `type C is V`, where `C` is the name of the newly introduced type and `V` has to be a built-in value type (the "underlying type"). The function `C.wrap` is used to convert from the underlying type to the custom type. Similarly, the function `C.unwrap` is used to convert from the custom type to the underlying type.

The type `C` does not have any operators or attached member functions. In particular, even the operator == is not defined. Explicit and implicit conversions to and from other types are disallowed.

The data-representation of values of such types are inherited from the underlying type and the underlying type is also used in the ABI.

The following example illustrates a custom type `UFixed256x18` representing a decimal fixed point type with 18 decimals and a minimal library to do arithmetic operations on the type.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// Represent a 18 decimal, 256 bit wide fixed point type using a user-defined value type.
type UFixed256x18 is uint256;

/// A minimal library to do fixed point operations on UFixed256x18.
library FixedMath {
    uint constant multiplier = 10**18;

    /// Adds two UFixed256x18 numbers. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
    }
    /// Multiplies UFixed256x18 and uint256. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }
    /// Take the floor of a UFixed256x18 number.
    /// @return the largest integer that does not exceed `a`.
    function floor(UFixed256x18 a) internal pure returns (uint256) {
        return UFixed256x18.unwrap(a) / multiplier;
    }
    /// Turns a uint256 into a UFixed256x18 of the same value.
    /// Reverts if the integer is too large.
    function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(a * multiplier);
    }
}
```

Notice how `UFixed256x18.wrap` and `FixedMath.toUFixed256x18` have the same signature but perform two very

different operations: The `UFixed256x18.wrap` function returns a `UFixed256x18` that has the same data representation as the input, whereas `toUFixed256x18` returns a `UFixed256x18` that has the same numerical value.

Function Types

Function types are the types of functions. Variables of a function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - *internal* and *external* functions:

Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Note that public functions of the current contract can be used both as an internal and as an external function. To use `f` as an internal function, just use `f`, if you want to use its external form, use `this.f`.

If a function type variable is not initialised, calling it results in a *Panic error*. The same happens if you call a function after using `delete` on it.

Note

Lambda or inline functions are planned but not yet supported.

Declaration syntax

Function types are notated as follows:

```
function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return
↪types>)]
```

In contrast to the parameter types, the return types cannot be empty - if the function type should not return anything, the whole `returns (<return types>)` part has to be omitted.

By default, function types are internal, so the `internal` keyword can be omitted. Note that this only applies to function types. Visibility has to be specified explicitly for functions defined in contracts, they do not have a default.

Conversions

A function type `A` is implicitly convertible to a function type `B` if and only if their parameter types are identical, their return types are identical, their internal/external property is identical and the state mutability of `A` is more restrictive than the state mutability of `B`. In particular:

- `pure` functions can be converted to `view` and `non-payable` functions
- `view` functions can be converted to `non-payable` functions
- `payable` functions can be converted to `non-payable` functions

No other conversions between function types are possible.

The rule about `payable` and `non-payable` might be a little confusing, but in essence, if a function is `payable`, this means that it also accepts a payment of zero Ether, so it also is `non-payable`. On the other hand, a `non-payable` function will reject Ether sent to it, so `non-payable` functions cannot be converted to `payable` functions. To clarify,

rejecting ether is more restrictive than not rejecting ether. This means you can override a payable function with a non-payable but not the other way around.

Additionally, When you define a `non-payable` function pointer, the compiler does not enforce that the pointed function will actually reject ether. Instead, it enforces that the function pointer is never used to send ether. Which makes it possible to assign a `payable` function pointer to a `non-payable` function pointer ensuring both types behave the same way, i.e, both cannot be used to send ether.

If external function types are used outside of the context of Solidity, they are treated as the `function` type, which encodes the address followed by the function identifier together in a single `bytes24` type.

A function of an internal type can be assigned to a variable of an internal function type regardless of where it is defined. This includes private, internal and public functions of both contracts and libraries as well as free functions. External function types, on the other hand, are only compatible with public and external contract functions.

Note

External functions with `calldata` parameters are incompatible with external function types with `calldata` parameters. They are compatible with the corresponding types with `memory` parameters instead. For example, there is no function that can be pointed at by a value of type `function (string calldata) external` while `function (string memory) external` can point at both `function f(string memory) external {}` and `function g(string calldata) external {}`. This is because for both locations the arguments are passed to the function in the same way. The caller cannot pass its calldata directly to an external function and always ABI-encodes the arguments into memory. Marking the parameters as `calldata` only affects the implementation of the external function and is meaningless in a function pointer on the caller's side.

Warning

Comparison of internal function pointers can have unexpected results in the legacy pipeline with the optimizer enabled, as it can collapse identical functions into one, which will then lead to said function pointers comparing as equal instead of not. Such comparisons are not advised, and will lead to the compiler issuing a warning, until the next breaking release (0.9.0), when the warning will be upgraded to an error, thereby making such comparisons disallowed.

Libraries are excluded because they require a `delegatecall` and use *a different ABI convention for their selectors*. Functions declared in interfaces do not have definitions so pointing at them does not make sense either.

Members

External (or public) functions have the following members:

- `.address` returns the address of the contract of the function.
- `.selector` returns the *ABI function selector*

Note

External (or public) functions used to have the additional members `.gas(uint)` and `.value(uint)`. These were deprecated in Solidity 0.6.2 and removed in Solidity 0.7.0. Instead use `{gas: ...}` and `{value: ...}` to specify the amount of gas or the amount of wei sent to a function, respectively. See *External Function Calls* for more information.

Value stability across contract updates

An important aspect to consider when using values of function types is whether the value will remain valid if the underlying code changes.

The state of the blockchain is not completely immutable and there are multiple ways to place different code under the same address:

- Directly deploying different code using *salted contract creation*.
- Delegating to a different contract via *DELEGATECALL* (upgradeable code behind a proxy contract is a common example of this).
- Account abstraction as defined by EIP-7702.

External function types can be considered as stable as contract's ABI, which makes them very portable. Their ABI representation always consists of a contract address and a function selector and it is perfectly safe to store them long-term or pass them between contracts. While it is possible for the referenced function to change or disappear, a direct external call would be affected the same way, so there is no additional risk in such use.

In case of internal functions, however, the value is an identifier that is strongly tied to contract's bytecode. The actual representation of the identifier is an implementation detail and may change between compiler versions or even *between different backends*. Values assigned under a given representation are deterministic (i.e. guaranteed to remain the same as long as the source code is the same) but are easily affected by changes such as adding, removing or reordering of functions. The compiler is also free to remove internal functions that are never used, which may affect other identifiers. Some representations, e.g. one where identifiers are simply jump targets, may be affected by virtually any change, even one completely unrelated to internal functions.

To counter this, the language limits the use of internal function types outside of the context in which they are valid. This is why internal function types cannot be used as parameters of external functions (or in any other way that is exposed in contract's ABI). However, there are still situations where it is up to the user to decide whether their use is safe or not. For example long-term storage of such values in state variables is discouraged, but may be safe if the contract code is never going to be updated. It is also always possible to side-step any safeguards by using inline assembly. Such use always needs careful consideration.

Note

The removal of unused internal functions only takes into account explicit references to such functions by name. Implicit references, such as assigning a new value to a function type variable in inline assembly may still lead to the removal of the function if it is not also referenced explicitly elsewhere in the source.

Examples

Example that shows how to use the members:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.4 <0.9.0;

contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
        return this.f.selector;
    }

    function g() public {
        this.f{gas: 10, value: 800}();
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Example that shows how to use internal function types:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library ArrayUtils {
    // internal functions can be used in internal library functions because
    // they will be part of the same code context
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }

    function reduce(
        uint[] memory self,
        function (uint, uint) pure returns (uint) f
    )
        internal
        pure
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }

    function range(uint length) internal pure returns (uint[] memory r) {
        r = new uint[](length);
        for (uint i = 0; i < r.length; i++) {
            r[i] = i;
        }
    }
}


contract Pyramid {
    using ArrayUtils for *;

    function pyramid(uint l) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }
```

(continues on next page)

(continued from previous page)

```
    function square(uint x) internal pure returns (uint) {
        return x * x;
    }

    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

Another example that uses external function types:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


contract Oracle {
    struct Request {
        bytes data;
        function(uint) external callback;
    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted source
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST =␣
↪Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa)); // known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
    }
```

(continues on next page)

(continued from previous page)

```
}
```

3.6.2 Reference Types

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored: `memory` (whose lifetime is limited to an external function call), `storage` (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or `calldata` (special data location that contains the function arguments).

An assignment or type conversion that changes the data location will always incur an automatic copy operation, while assignments inside the same data location only copy in some cases for storage types.

Data location

Every reference type has an additional annotation, the "data location", about where it is stored. There are three data locations: `memory`, `storage` and `calldata`. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory.

Note

`transient` is not yet supported as a data location for reference types.

Note

If you can, try to use `calldata` as data location because it will avoid copies and also makes sure that the data cannot be modified. Arrays and structs with `calldata` data location can also be returned from functions, but it is not possible to allocate such types.

Note

Arrays and structs with `calldata` location declared in a function body or as its return parameters must be assigned before being used or returned. There are certain cases in which non-trivial control flow is used and the compiler can't properly detect the initialization. A common workaround in such cases is to assign the affected variable to itself before the correct initialization takes place.

Note

Prior to version 0.6.9 data location for reference-type arguments was limited to `calldata` in external functions, `memory` in public functions and either `memory` or `storage` in internal and private ones. Now `memory` and `calldata` are allowed in all functions regardless of their visibility.

Note

Constructor parameters cannot use `calldata` as their data location.

> **Note**
>
> Prior to version 0.5.0 the data location could be omitted, and would default to different locations depending on the kind of variable, function type, etc., but all complex types must now give an explicit data location.

Data location and assignment behavior

Data locations are not only relevant for persistency of data, but also for the semantics of assignments:

- Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a **local** storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // Similarly, "delete y" is not valid, as assignments to local variables
        // referencing storage objects can only be made from existing storage objects.
        // It would "reset" the pointer, but there is no sensible location it could␣
↪point to.
        // For more details see the documentation of the "delete" operator.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

Arrays

Arrays can have a compile-time fixed size, or they can have a dynamic size.

The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.

For example, an array of 5 dynamic arrays of `uint` is written as `uint[][5]`. The notation is reversed compared to some other languages. In Solidity, `X[3]` is always an array containing three elements of type `X`, even if `X` is itself an array. This is not the case in other languages such as C.

Indices are zero-based, and access is in the opposite direction of the declaration.

For example, if you have a variable `uint[][5] memory x`, you access the seventh `uint` in the third dynamic array using `x[2][6]`, and to access the third dynamic array, use `x[2]`. Again, if you have an array `T[5] a` for a type `T` that can also be an array, then `a[2]` always has type `T`.

Array elements can be of any type, including mapping or struct. The general restrictions for types apply, in that mappings can only be stored in the `storage` data location and publicly-visible functions need parameters that are *ABI types*.

It is possible to mark state variable arrays `public` and have Solidity create a *getter*. The numeric index becomes a required parameter for the getter.

Accessing an array past its end causes a failing assertion. Methods `.push()` and `.push(value)` can be used to append a new element at the end of a dynamically-sized array, where `.push()` appends a zero-initialized element and returns a reference to it.

> **Note**
>
> Dynamically-sized arrays can only be resized in storage. In memory, such arrays can be of arbitrary size but the size cannot be changed once an array is allocated.

`bytes` and `string` as Arrays

Variables of type `bytes` and `string` are special arrays. The `bytes` type is similar to `bytes1[]`, but it is packed tightly in calldata and memory. `string` is equal to `bytes` but does not allow length or index access.

Solidity does not have string manipulation functions, but there are third-party string libraries. You can also compare two strings by their keccak256-hash using `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` and concatenate two strings using `string.concat(s1, s2)`.

You should use `bytes` over `bytes1[]` because it is cheaper, since using `bytes1[]` in `memory` adds 31 padding bytes between the elements. Note that in `storage`, the padding is absent due to tight packing, see *bytes and string*. As a general rule, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of the value types `bytes1` to `bytes32` because they are much cheaper.

> **Note**
>
> If you want to access the byte-representation of a string `s`, use `bytes(s).length` / `bytes(s)[7] = 'x';`. Keep in mind that you are accessing the low-level bytes of the UTF-8 representation, and not the individual characters.

The functions `bytes.concat` and `string.concat`

You can concatenate an arbitrary number of `string` values using `string.concat`. The function returns a single `string memory` array that contains the contents of the arguments without padding. If you want to use parameters of other types that are not implicitly convertible to `string`, you need to convert them to `string` first.

Analogously, the `bytes.concat` function can concatenate an arbitrary number of `bytes` or `bytes1 ... bytes32` values. The function returns a single `bytes memory` array that contains the contents of the arguments without padding. If you want to use string parameters or other types that are not implicitly convertible to `bytes`, you need to convert them to `bytes` or `bytes1`/.../`bytes32` first.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.12;

contract C {
    string s = "Storage";
    function f(bytes calldata bc, string memory sm, bytes16 b) public view {
        string memory concatString = string.concat(s, string(bc), "Literal", sm);
        assert((bytes(s).length + bc.length + 7 + bytes(sm).length) ==␣
↪bytes(concatString).length);

        bytes memory concatBytes = bytes.concat(bytes(s), bc, bc[:2], "Literal",␣
↪bytes(sm), b);
        assert((bytes(s).length + bc.length + 2 + 7 + bytes(sm).length + b.length) ==␣
↪concatBytes.length);
    }
}
```

If you call `string.concat` or `bytes.concat` without arguments they return an empty array.

Allocating Memory Arrays

Memory arrays with dynamic length can be created using the `new` operator. As opposed to storage arrays, it is **not** possible to resize memory arrays (e.g. the `.push` member functions are not available). You either have to calculate the required size in advance or create a new memory array and copy every element.

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the *default value*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

Array Literals

An array literal is a comma-separated list of one or more expressions, enclosed in square brackets (`[...]`). For example `[1, a, f(3)]`. The type of the array literal is determined as follows:

It is always a statically-sized memory array whose length is the number of expressions.

The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a type error if this is not possible.

It is not enough that there is a type all the elements can be converted to. One of the elements has to be of that type.

In the example below, the type of `[1, 2, 3]` is `uint8[3] memory`, because the type of each of these constants is `uint8`. If you want the result to be a `uint[3] memory` type, you need to convert the first element to `uint`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] memory) public pure {
        // ...
    }
}
```

The array literal `[1, -1]` is invalid because the type of the first expression is `uint8` while the type of the second is `int8` and they cannot be implicitly converted to each other. To make it work, you can use `[int8(1), -1]`, for example.

Since fixed-size memory arrays of different type cannot be converted into each other (even if the base types can), you always have to specify a common base type explicitly if you want to use two-dimensional array literals:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffff, 2], [uint24(0xff), 3],␣
↪[uint24(0xffff), 4]];
        // The following does not work, because some of the inner arrays are not of the␣
↪right type.
        // uint[2][4] memory x = [[0x1, 1], [0xffffff, 2], [0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

// This will not compile.
contract C {
    function f() public {
```

(continues on next page)

(continued from previous page)

```
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

It is planned to remove this restriction in the future, but it creates some complications because of how arrays are passed in the ABI.

If you want to initialize dynamically-sized arrays, you have to assign the individual elements:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        uint[] memory x = new uint[](3);
        x[0] = 1;
        x[1] = 3;
        x[2] = 4;
    }
}
```

Array Members

length:
Arrays have a `length` member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

push():
Dynamic storage arrays and `bytes` (not `string`) have a member function called `push()` that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like `x.push().t = 2` or `x.push() = b`.

push(x):
Dynamic storage arrays and `bytes` (not `string`) have a member function called `push(x)` that you can use to append a given element at the end of the array. The function returns nothing.

pop():
Dynamic storage arrays and `bytes` (not `string`) have a member function called `pop()` that you can use to remove an element from the end of the array. This also implicitly calls *delete* on the removed element. The function returns nothing.

Note

Increasing the length of a storage array by calling `push()` has constant gas costs because storage is zero-initialised, while decreasing the length by calling `pop()` has a cost that depends on the "size" of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling *delete* on them.

Note

To use arrays of arrays in external (instead of public) functions, you need to activate ABI coder v2.

Note

In EVM versions before Byzantium, it was not possible to access dynamic arrays returned from function calls. If you call functions that return dynamic arrays, make sure to use an EVM that is set to Byzantium mode.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract ArrayContract {
    uint[2**20] aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    // In Solidity, T[k] and T[] are always arrays with elements of type T,
    // even if T itself is an array.
    // Because of that, bool[2][] is a dynamic array of elements
    // that are bool[2]. This is different from other languages, like C.
    // Data location for all state variables is storage.
    bool[2][] pairsOfFlags;

    // newPairs is stored in memory
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // assignment to a storage array performs a copy of ``newPairs`` and
        // replaces the complete array ``pairsOfFlags``.
        pairsOfFlags = newPairs;
    }

    struct StructType {
        uint[] contents;
        uint moreInfo;
    }
    StructType s;

    function f(uint[] memory c) public {
        // stores a reference to ``s`` in ``g``
        StructType storage g = s;
        // also changes ``s.moreInfo``.
        g.moreInfo = 2;
        // assigns a copy because ``g.contents``
        // is not a local variable, but a member of
        // a local variable.
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // access to a non-existing index will throw an exception
        pairsOfFlags[index][0] = flagA;
        pairsOfFlags[index][1] = flagB;
    }
```

(continues on next page)

(continued from previous page)

```
    function changeFlagArraySize(uint newSize) public {
        // using push and pop is the only way to change the
        // length of an array
        if (newSize < pairsOfFlags.length) {
            while (pairsOfFlags.length > newSize)
                pairsOfFlags.pop();
        } else if (newSize > pairsOfFlags.length) {
            while (pairsOfFlags.length < newSize)
                pairsOfFlags.push();
        }
    }

    function clear() public {
        // these clear the arrays completely
        delete pairsOfFlags;
        delete aLotOfIntegers;
        // identical effect here
        pairsOfFlags = new bool[2][](0);
    }

    bytes byteData;

    function byteArrays(bytes memory data) public {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        byteData = data;
        for (uint i = 0; i < 7; i++)
            byteData.push();
        byteData[3] = 0x08;
        delete byteData[2];
    }

    function addFlag(bool[2] memory flag) public returns (uint) {
        pairsOfFlags.push(flag);
        return pairsOfFlags.length;
    }

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Inline arrays are always statically-sized and if you only
        // use literals, you have to provide at least one type.
        arrayOfPairs[0] = [uint(1), 2];

        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = bytes1(uint8(i));
        return b;
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Dangling References to Storage Array Elements

When working with storage arrays, you need to take care to avoid dangling references. A dangling reference is a reference that points to something that no longer exists or has been moved without updating the reference. A dangling reference can for example occur, if you store a reference to an array element in a local variable and then `.pop()` from the containing array:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[][] s;

    function f() public {
        // Stores a pointer to the last array element of s.
        uint[] storage ptr = s[s.length - 1];
        // Removes the last array element of s.
        s.pop();
        // Writes to the array element that is no longer within the array.
        ptr.push(0x42);
        // Adding a new element to ``s`` now will not add an empty array, but
        // will result in an array of length 1 with ``0x42`` as element.
        s.push();
        assert(s[s.length - 1][0] == 0x42);
    }
}
```

The write in `ptr.push(0x42)` will **not** revert, despite the fact that `ptr` no longer refers to a valid element of `s`. Since the compiler assumes that unused storage is always zeroed, a subsequent `s.push()` will not explicitly write zeroes to storage, so the last element of `s` after that `push()` will have length 1 and contain `0x42` as its first element.

Note that Solidity does not allow to declare references to value types in storage. These kinds of explicit dangling references are restricted to nested reference types. However, dangling references can also occur temporarily when using complex expressions in tuple assignments:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[] s;
    uint[] t;
    constructor() {
        // Push some initial values to the storage arrays.
        s.push(0x07);
        t.push(0x03);
    }

    function g() internal returns (uint[] storage) {
        s.pop();
        return t;
```

(continues on next page)

(continued from previous page)

```
    }

    function f() public returns (uint[] memory) {
        // The following will first evaluate `s.push()` to a reference to a new element
        // at index 1. Afterwards, the call to `g` pops this new element, resulting in
        // the left-most tuple element to become a dangling reference. The assignment␣
↪still
        // takes place and will write outside the data area of `s`.
        (s.push(), g()[0]) = (0x42, 0x17);
        // A subsequent push to `s` will reveal the value written by the previous
        // statement, i.e. the last element of `s` at the end of this function will have
        // the value `0x42`.
        s.push();
        return s;
    }
}
```

It is always safer to only assign to storage once per statement and to avoid complex expressions on the left-hand-side of an assignment.

You need to take particular care when dealing with references to elements of `bytes` arrays, since a `.push()` on a bytes array may switch *from short to long layout in storage*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

// This will report a warning
contract C {
    bytes x = "0123456789012345678901234567890";

    function test() external returns(uint) {
        (x.push(), x.push()) = (0x01, 0x02);
        return x.length;
    }
}
```

Here, when the first `x.push()` is evaluated, `x` is still stored in short layout, thereby `x.push()` returns a reference to an element in the first storage slot of `x`. However, the second `x.push()` switches the bytes array to large layout. Now the element that `x.push()` referred to is in the data area of the array while the reference still points at its original location, which is now a part of the length field and the assignment will effectively garble the length of `x`. To be safe, only enlarge bytes arrays by at most one element during a single assignment and do not simultaneously index-access the array in the same statement.

While the above describes the behavior of dangling storage references in the current version of the compiler, any code with dangling references should be considered to have *undefined behavior*. In particular, this means that any future version of the compiler may change the behavior of code that involves dangling references.

Be sure to avoid dangling references in your code!

Array Slices

Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a uint256 type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

If `start` is greater than `end` or if `end` is greater than the length of the array, an exception is thrown.

Both `start` and `end` are optional: `start` defaults to `0` and `end` defaults to the length of the array.

Array slices do not have any members. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.

Array slices do not have a type name which means no variable can have an array slices as type, they only exist in intermediate expressions.

> **Note**
>
> As of now, array slices are only implemented for calldata arrays.

Array slices are useful to ABI-decode secondary data passed in function parameters:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.5 <0.9.0;
contract Proxy {
    /// @dev Address of the client contract managed by proxy i.e., this contract
    address client;

    constructor(address client_) {
        client = client_;
    }

    /// Forward call to "setOwner(address)" that is implemented by client
    /// after doing basic validation on the address argument.
    function forward(bytes calldata payload) external {
        bytes4 sig = bytes4(payload[:4]);
        // Due to truncating behavior, bytes4(payload) performs identically.
        // bytes4 sig = bytes4(payload);
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(payload);
        require(status, "Forwarded call failed.");
    }
}
```

Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// Defines a new type with two fields.
// Declaring a struct outside of a contract allows
// it to be shared by multiple contracts.
// Here, this is not really needed.
struct Funder {
    address addr;
```

(continues on next page)

(continued from previous page)

```
    uint amount;
}

contract CrowdFunding {
    // Structs can also be defined inside contracts, which makes them
    // visible only there and in derived contracts.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping(uint => Funder) funders;
    }

    uint numCampaigns;
    mapping(uint => Campaign) campaigns;

    function newCampaign(address payable beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // We cannot use "campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)"
        // because the right hand side creates a memory-struct "Campaign" that contains␣
↪a mapping.
        Campaign storage c = campaigns[campaignID];
        c.beneficiary = beneficiary;
        c.fundingGoal = goal;
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // Creates a new temporary memory struct, initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender, msg.value) to initialise.
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        (bool success, ) = c.beneficiary.call{value: amount}("");
        return success;
    }
}
```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a

mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite.

Note how in all the functions, a struct type is assigned to a local variable with data location `storage`. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

Of course, you can also directly access the members of the struct without assigning it to a local variable, as in `campaigns[campaignID].amount = 0`.

Note

Until Solidity 0.7.0, memory-structs containing members of storage-only types (e.g. mappings) were allowed and assignments like `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` in the example above would work and just silently skip those members.

3.6.3 Mapping Types

Mapping types use the syntax `mapping(KeyType KeyName? => ValueType ValueName?)` and variables of mapping type are declared using the syntax `mapping(KeyType KeyName? => ValueType ValueName?) VariableName`. The `KeyType` can be any built-in value type, `bytes`, `string`, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. `ValueType` can be any type, including mappings, arrays and structs. `KeyName` and `ValueName` are optional (so `mapping(KeyType => ValueType)` works as well) and can be any valid identifier that is not a type.

You can think of mappings as hash tables, which are virtually initialised such that every possible key exists and is mapped to a value whose byte-representation is all zeros, a type's *default value*. The similarity ends there, the key data is not stored in a mapping, only its `keccak256` hash is used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information regarding the assigned keys (see *Clearing Mappings*).

Mappings can only have a data location of `storage` and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.

You can mark state variables of mapping type as `public` and Solidity creates a *getter* for you. The `KeyType` becomes a parameter with name `KeyName` (if specified) for the getter. If `ValueType` is a value type or a struct, the getter returns `ValueType` with name `ValueName` (if specified). If `ValueType` is an array or a mapping, the getter has one parameter for each `KeyType`, recursively.

In the example below, the `MappingExample` contract defines a public `balances` mapping, with the key type an `address`, and a value type a `uint`, mapping an Ethereum address to an unsigned integer value. As `uint` is a value type, the getter returns a value that matches the type, which you can see in the `MappingUser` contract that returns the value at the specified address.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
```

(continues on next page)

(continued from previous page)

```
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

The example below is a simplified version of an ERC20 token. `_allowances` is an example of a mapping type inside another mapping type.

In the example below, the optional `KeyName` and `ValueName` are provided for the mapping. It does not affect any contract functionality or bytecode, it only sets the `name` field for the inputs and outputs in the ABI for the mapping's getter.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;

contract MappingExampleWithNames {
    mapping(address user => uint balance) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}
```

The example below uses `_allowances` to record the amount someone else is allowed to withdraw from your account.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract MappingExample {

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public␣
↪returns (bool) {
        require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high␣
↪enough.");
        _allowances[sender][msg.sender] -= amount;
        _transfer(sender, recipient, amount);
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
    function approve(address spender, uint256 amount) public returns (bool) {
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
        require(_balances[sender] >= amount, "ERC20: Not enough funds.");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

Iterable Mappings

You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that. For example, the code below implements an `IterableMapping` library that the `User` contract then adds data to, and the `sum` function iterates over to sum all the values.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

type Iterator is uint;

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool␣
↪replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
```

(continues on next page)

(continued from previous page)

```
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterateStart(itmap storage self) internal view returns (Iterator) {
        return iteratorSkipDeleted(self, 0);
    }

    function iterateValid(itmap storage self, Iterator iterator) internal view returns␣
↪(bool) {
        return Iterator.unwrap(iterator) < self.keys.length;
    }

    function iterateNext(itmap storage self, Iterator iterator) internal view returns␣
↪(Iterator) {
        return iteratorSkipDeleted(self, Iterator.unwrap(iterator) + 1);
    }

    function iterateGet(itmap storage self, Iterator iterator) internal view returns␣
↪(uint key, uint value) {
        uint keyIndex = Iterator.unwrap(iterator);
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }

    function iteratorSkipDeleted(itmap storage self, uint keyIndex) private view returns␣
↪(Iterator) {
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return Iterator.wrap(keyIndex);
    }
}

// How to use it
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
```

(continues on next page)

(continued from previous page)

```
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            Iterator i = data.iterateStart();
            data.iterateValid(i);
            i = data.iterateNext(i)
        ) {
            (, uint value) = data.iterateGet(i);
            s += value;
        }
    }
}
```

3.6.4 Operators

Arithmetic and bit operators can be applied even if the two operands do not have the same type. For example, you can compute `y = x + z`, where `x` is a `uint8` and `z` has the type `uint32`. In these cases, the following mechanism will be used to determine the type in which the operation is computed (this is important in case of overflow) and the type of the operator's result:

1. If the type of the right operand can be implicitly converted to the type of the left operand, use the type of the left operand,
2. if the type of the left operand can be implicitly converted to the type of the right operand, use the type of the right operand,
3. otherwise, the operation is not allowed.

In case one of the operands is a *literal number* it is first converted to its "mobile type", which is the smallest type that can hold the value (unsigned types of the same bit-width are considered "smaller" than the signed types). If both are literal numbers, the operation is computed with effectively unlimited precision in that the expression is evaluated to whatever precision is necessary so that none is lost when the result is used with a non-literal type.

The operator's result type is the same as the type the operation is performed in, except for comparison operators where the result is always `bool`.

The operators ** (exponentiation), << and >> use the type of the left operand for the operation and the result.

Ternary Operator

The ternary operator is used in expressions of the form `<expression> ? <trueExpression> : <falseExpression>`. It evaluates one of the latter two given expressions depending upon the result of the evaluation of the main `<expression>`. If `<expression>` evaluates to `true`, then `<trueExpression>` will be evaluated, otherwise `<falseExpression>` is evaluated.

The result of the ternary operator does not have a rational number type, even if all of its operands are rational number literals. The result type is determined from the types of the two operands in the same way as above, converting to their mobile type first if required.

As a consequence, `255 + (true ? 1 : 0)` will revert due to arithmetic overflow. The reason is that `(true ? 1 : 0)` is of `uint8` type, which forces the addition to be performed in `uint8` as well, and 256 exceeds the range allowed for this type.

Another consequence is that an expression like `1.5 + 1.5` is valid but `1.5 + (true ? 1.5 : 2.5)` is not. This is because the former is a rational expression evaluated in unlimited precision and only its final value matters. The latter involves a conversion of a fractional rational number to an integer, which is currently disallowed.

Compound and Increment/Decrement Operators

If `a` is an LValue (i.e. a variable or something that can be assigned to), the following operators are available as shorthands:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=` and `>>=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1 / a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched. This especially means that it leaves a gap in the array. If you plan to remove items, a *mapping* is probably a better choice.

For structs, it assigns a struct with all members reset. In other words, the value of `a` after `delete a` is the same as if `a` would be declared without assignment, with the following caveat:

`delete` has no effect on mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings. However, individual keys and what they map to can be deleted: If `a` is a mapping, then `delete a[x]` will delete the value stored at `x`.

It is important to note that `delete a` really behaves like an assignment to `a`, i.e. it stores a new object in `a`. This distinction is visible when `a` is reference variable: It will only reset `a` itself, not the value it referred to previously.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a␣
↪complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local variables
        // referencing storage objects can only be made from existing storage objects.
```

(continues on next page)

(continued from previous page)

```
        assert(y.length == 0);
    }
}
```

Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	`++`, `--`
	New expression	`new <typename>`
	Array subscripting	`<array>[<index>]`
	Member access	`<object>.<member>`
	Function-like call	`<func>(<args...>)`
	Parentheses	`(<statement>)`
2	Prefix increment and decrement	`++`, `--`
	Unary minus	`-`
	Unary operations	`delete`
	Logical NOT	`!`
	Bitwise NOT	`~`
3	Exponentiation	`**`
4	Multiplication, division and modulo	`*`, `/`, `%`
5	Addition and subtraction	`+`, `-`
6	Bitwise shift operators	`<<`, `>>`
7	Bitwise AND	`&`
8	Bitwise XOR	`^`
9	Bitwise OR	`|`
10	Inequality operators	`<`, `>`, `<=`, `>=`
11	Equality operators	`==`, `!=`
12	Logical AND	`&&`
13	Logical OR	`||`
14	Ternary operator	`<conditional> ? <if-true> : <if-false>`
	Assignment operators	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
15	Comma operator	`,`

3.6.5 Conversions between Elementary Types

Implicit Conversions

An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

For example, `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256`, because `uint256` cannot hold values such as `-1`.

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). This means that operations are always performed in the type of one of the operands.

For more details about which implicit conversions are possible, please consult the sections about the types themselves.

In the example below, `y` and `z`, the operands of the addition, do not have the same type, but `uint8` can be implicitly converted to `uint16` and not vice-versa. Because of that, `y` is converted to the type of `z` before the addition is performed in the `uint16` type. The resulting type of the expression `y + z` is `uint16`. Because it is assigned to a variable of type `uint32` another implicit conversion is performed after the addition.

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

Explicit Conversions

If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behavior and allows you to bypass some security features of the compiler, so be sure to test that the result is what you want and expect!

Take the following example that converts a negative `int` to a `uint`:

```
int  y = -3;
uint x = uint(y);
```

At the end of this code snippet, `x` will have the value `0xfffff..fd` (64 hex characters), which is -3 in the two's complement representation of 256 bits.

If an integer is explicitly converted to a smaller type, higher-order bits are cut off:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end). The result of the conversion will compare equal to the original integer:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

Fixed-size bytes types behave differently during conversions. They can be thought of as sequences of individual bytes and converting to a smaller type will cut off the sequence:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b will be 0x12
```

If a fixed-size bytes type is explicitly converted to a larger type, it is padded on the right. Accessing the byte at a fixed index will result in the same value before and after the conversion (if the index is still in range):

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b will be 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Since integers and fixed-size byte arrays behave differently when truncating or padding, explicit conversions between integers and fixed-size byte arrays are only allowed, if both have the same size. If you want to convert between integers and fixed-size byte arrays of different size, you have to use intermediate conversions that make the desired truncation and padding rules explicit:

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b will be 0x00001234
uint32 c = uint32(bytes4(a)); // c will be 0x12340000
uint8 d = uint8(uint16(a)); // d will be 0x34
uint8 e = uint8(bytes1(a)); // e will be 0x12
```

`bytes` arrays and `bytes` calldata slices can be converted explicitly to fixed bytes types (`bytes1`/.../`bytes32`). In case the array is longer than the target fixed bytes type, truncation at the end will happen. If the array is shorter than the target type, it will be padded with zeros at the end.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16, bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m);  // if length of m is greater than 16, truncation will␣
↪happen
        b = bytes16(s);  // padded on the right, so result is "abcdefgh\0\0\0\0\0\0\0\0"
        bytes3 b1 = bytes3(s); // truncated, b1 equals to "abc"
        b = bytes16(c[:8]);  // also padded with zeros
        return (b, b1);
    }
}
```

3.6.6 Conversions between Literals and Elementary Types

Integer Types

Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation:

```
uint8 a = 12; // fine
uint32 b = 1234; // fine
uint16 c = 0x123456; // fails, since it would have to truncate to 0x3456
```

Note

Prior to version 0.8.0, any decimal or hexadecimal number literals could be explicitly converted to an integer type. From 0.8.0, such explicit conversions are as strict as implicit conversions, i.e., they are only allowed if the literal fits in the resulting range.

Fixed-Size Byte Arrays

Decimal number literals cannot be implicitly converted to fixed-size byte arrays. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the bytes type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type:

```
bytes2 a = 54321; // not allowed
bytes2 b = 0x12; // not allowed
bytes2 c = 0x123; // not allowed
```

(continues on next page)

(continued from previous page)

```
bytes2 d = 0x1234; // fine
bytes2 e = 0x0012; // fine
bytes4 f = 0; // fine
bytes4 g = 0x0; // fine
```

String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters is less than or equal to the size of the bytes type:

```
bytes2 a = hex"1234"; // fine
bytes2 b = "xy"; // fine
bytes2 c = hex"12"; // fine
bytes2 e = "x"; // fine
bytes2 f = "xyz"; // not allowed
```

Addresses

As described in *Address Literals*, hex literals of the correct size that pass the checksum test are of `address` type. No other literals can be implicitly converted to the `address` type.

Explicit conversions to `address` are allowed only from `bytes20` and `uint160`.

An `address a` can be converted explicitly to `address payable` via `payable(a)`.

Note

Prior to version 0.8.0, it was possible to explicitly convert from any integer type (of any size, signed or unsigned) to `address` or `address payable`. Starting with 0.8.0 only conversion from `uint160` is allowed.

3.7 Units and Globally Available Variables

3.7.1 Ether Units

A literal number can take a suffix of `wei`, `gwei` or `ether` to specify a subdenomination of Ether, where Ether numbers without a postfix are assumed to be Wei.

```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);
```

The only effect of the subdenomination suffix is a multiplication by a power of ten.

Note

The denominations `finney` and `szabo` have been removed in version 0.7.0.

3.7.2 Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days` and `weeks` after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`

- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds. Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

Note

The suffix `years` has been removed in version 0.5.0 due to the reasons above.

These suffixes cannot be applied to variables. For example, if you want to interpret a function parameter in days, you can in the following way:

```
function f(uint start, uint daysAfter) public {
    if (block.timestamp >= start + daysAfter * 1 days) {
        // ...
    }
}
```

3.7.3 Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

Block and Transaction Properties

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block when `blocknumber` is one of the 256 most recent blocks; otherwise returns zero
- `blobhash(uint index) returns (bytes32)`: versioned hash of the `index`-th blob associated with the current transaction. A versioned hash consists of a single byte representing the version (currently `0x01`), followed by the last 31 bytes of the SHA256 hash of the KZG commitment (EIP-4844). Returns zero if no blob with the given index exists.
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.blobbasefee` (`uint`): current block's blob base fee (EIP-7516 and EIP-4844)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty (`EVM < Paris`). For other EVM versions it behaves as a deprecated alias for `block.prevrandao` (EIP-4399)
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.prevrandao` (`uint`): random number provided by the beacon chain (`EVM >= Paris`)
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas

- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Note

The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every **external** function call. This includes calls to library functions.

Note

When contracts are evaluated off-chain rather than in context of a transaction included in a block, you should not assume that `block.*` and `tx.*` refer to values from any specific block or transaction. These values are provided by the EVM implementation that executes the contract and can be arbitrary.

Note

Do not rely on `block.timestamp` or `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any compensation, e.g. Ether.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note

The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Note

The function `blockhash` was previously known as `block.blockhash`, which was deprecated in version 0.4.22 and removed in version 0.5.0.

Note

The function `gasleft` was previously known as `msg.gas`, which was deprecated in version 0.4.21 and removed in version 0.5.0.

Note

In version 0.7.0, the alias `now` (for `block.timestamp`) was removed.

ABI Encoding and Decoding Functions

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI-decodes the given data, while the types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: ABI-encodes the given arguments
- `abi.encodePacked(...) returns (bytes memory)`: Performs *packed encoding* of the given arguments. Note that packed encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`: ABI-encodes a call to `functionPointer` with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals `abi.encodeWithSelector(functionPointer.selector, (...))`

Note

These encoding functions can be used to craft data for external function calls without actually calling an external function. Furthermore, `keccak256(abi.encodePacked(a, b))` is a way to compute the hash of structured data (although be aware that it is possible to craft a "hash collision" using different function parameter types).

See the documentation about the *ABI* and the *tightly packed encoding* for details about the encoding.

Members of bytes

- `bytes.concat(...) returns (bytes memory)`: *Concatenates variable number of bytes and bytes1, ..., bytes32 arguments to one byte array*

Members of string

- `string.concat(...) returns (string memory)`: *Concatenates variable number of string arguments to one string array*

Error Handling

See the dedicated section on *assert and require* for more details on error handling and when to use which function.

`assert(bool condition)`
causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.

`require(bool condition)`
reverts if the condition is not met - to be used for errors in inputs or external components.

`require(bool condition, string memory message)`
reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.

`revert()`
: abort execution and revert state changes

`revert(string memory reason)`
: abort execution and revert state changes, providing an explanatory string

Mathematical and Cryptographic Functions

`addmod(uint x, uint y, uint k) returns (uint)`
: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

`mulmod(uint x, uint y, uint k) returns (uint)`
: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

`keccak256(bytes memory) returns (bytes32)`
: compute the Keccak-256 hash of the input

Note

There used to be an alias for `keccak256` called `sha3`, which was removed in version 0.5.0.

`sha256(bytes memory) returns (bytes32)`
: compute the SHA-256 hash of the input

`ripemd160(bytes memory) returns (bytes20)`
: compute RIPEMD-160 hash of the input

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`
: recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature:

  - `r` = first 32 bytes of signature
  - `s` = second 32 bytes of signature
  - `v` = final 1 byte of signature

  `ecrecover` returns an `address`, and not an `address payable`. See *address payable* for conversion, in case you need to transfer funds to the recovered address.

  For further details, read example usage.

Warning

If you use `ecrecover`, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. In the Homestead hard fork, this issue was fixed for _transaction_ signatures (see EIP-2), but the ecrecover function remained unchanged.

This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin has an ECDSA helper library that you can use as a wrapper for `ecrecover` without this issue.

Note

When running `sha256`, `ripemd160` or `ecrecover` on a *private blockchain*, you might encounter Out-of-Gas. This is because these functions are implemented as "precompiled contracts" and only really exist after they receive the

first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution might run into an Out-of-Gas error. A workaround for this problem is to first send Wei (1 for example) to each of the contracts before you use them in your actual contracts. This is not an issue on the main or test net.

`erc7201(string memory id) returns (uint)`
compute the base slot of a storage namespace of a given `id` according to the `erc7201` formula defined by ERC-7201. The formula is equivalent to `keccak256(keccak256(id) - 1) & ~0xff`. The builtin accepts arbitrary strings, including ones containing whitespace. The function can be used in compile-time context.

Members of Address Types

These members are explained in more detail in the section on *members of address*.

`<address>.balance` (`uint256`)
balance of the *Address* in Wei

`<address>.code` (`bytes memory`)
code at the *Address* (can be empty)

`<address>.codehash` (`bytes32`)
the codehash of the *Address*

`<address payable>.transfer(uint256 amount)`
send given amount of Wei to *Address*, reverts on failure, forwards 2300 gas stipend, not adjustable

`<address payable>.send(uint256 amount) returns (bool)`
send given amount of Wei to *Address*, returns `false` on failure, forwards 2300 gas stipend, not adjustable

Warning

`send()` and `transfer()` are deprecated and scheduled for removal. See the section on *send* and *transfer* for more information.

`<address>.call(bytes memory) returns (bool, bytes memory)`
issue low-level `CALL` with the given payload, returns success condition and return data, forwards all available gas (subject to additional limits imposed by some EVM versions), adjustable

`<address>.delegatecall(bytes memory) returns (bool, bytes memory)`
issue low-level `DELEGATECALL` with the given payload, returns success condition and return data, forwards all available gas (subject to additional limits imposed by some EVM versions), adjustable

`<address>.staticcall(bytes memory) returns (bool, bytes memory)`
issue low-level `STATICCALL` with the given payload, returns success condition and return data, forwards all available gas (subject to additional limits imposed by some EVM versions), adjustable

For more information, see the section on *Address*.

Warning

You should avoid using `.call()` whenever possible when executing another contract function as it bypasses type checking, function existence check, and argument packing.

Warning

There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: Use a pattern where the recipient withdraws the Ether.

Warning

Due to the fact that the EVM considers a call to a non-existing contract to always succeed, Solidity includes an extra check using the `extcodesize` opcode when performing external calls. This ensures that the contract that is about to be called either actually exists (it contains code) or an exception is raised.

The low-level calls which operate on addresses rather than contract instances (i.e. `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` and `.transfer()`) **do not** include this check, which makes them cheaper in terms of gas but also less safe.

Note

Prior to version 0.5.0, Solidity allowed address members to be accessed by a contract instance, for example `this.balance`. This is now forbidden and an explicit conversion to address must be done: `address(this).balance`.

Note

If state variables are accessed via a low-level delegatecall, the storage layout of the two contracts must align in order for the called contract to correctly access the storage variables of the calling contract by name. This is of course not the case if storage pointers are passed as function arguments as in the case for the high-level libraries.

Note

Prior to version 0.5.0, `.call`, `.delegatecall` and `.staticcall` only returned the success condition and not the return data.

Note

Prior to version 0.5.0, there was a member called `callcode` with similar but slightly different semantics than `delegatecall`.

Contract-related

`this` (current contract's type)
: The current contract, explicitly convertible to *Address*

`super`
: A contract one level higher in the inheritance hierarchy

`selfdestruct(address payable recipient)`
: Destroy the current contract, sending its funds to the given *Address* and end execution. Note that `selfdestruct`

has some peculiarities inherited from the EVM:

- the receiving contract's receive function is not executed.
- the contract is only really destroyed at the end of the transaction and `revert` s might "undo" the destruction.

Furthermore, all functions of the current contract are callable directly including the current function.

Warning

From `EVM >= Cancun` onwards, `selfdestruct` will **only** send all Ether in the account to the given recipient and not destroy the contract. However, when `selfdestruct` is called in the same transaction that creates the contract calling it, the behaviour of `selfdestruct` before Cancun hardfork (i.e., `EVM <= Shanghai`) is preserved and will destroy the current contract, deleting any data, including storage keys, code and the account itself. See EIP-6780 for more details.

The new behaviour is the result of a network-wide change that affects all contracts present on the Ethereum mainnet and testnets. It is important to note that this change is dependent on the EVM version of the chain on which the contract is deployed. The `--evm-version` setting used when compiling the contract has no bearing on it.

Also, note that the `selfdestruct` opcode has been deprecated in Solidity version 0.8.18, as recommended by EIP-6049. The deprecation is still in effect and the compiler will still emit warnings on its use. Any use in newly deployed contracts is strongly discouraged even if the new behavior is taken into account. Future changes to the EVM might further reduce the functionality of the opcode.

Note

Prior to version 0.5.0, there was a function called `suicide` with the same semantics as `selfdestruct`.

Type Information

The expression `type(X)` can be used to retrieve information about the type `X`. Currently, there is limited support for this feature (`X` can be either a contract or an integer type) but it might be expanded in the future.

The following properties are available for a contract type `C`:

`type(C).name`
The name of the contract.

`type(C).creationCode`
Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the `create2` opcode. This property can **not** be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.

`type(C).runtimeCode`
Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of `C`. If `C` has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with `.creationCode` also apply for this property.

In addition to the properties above, the following properties are available for an interface type `I`:

`type(I).interfaceId`
A `bytes4` value containing the EIP-165 interface identifier of the given interface `I`. This identifier is defined as the `XOR` of all function selectors defined within the interface itself - excluding all inherited functions.

The following properties are available for an integer type T:

type(T).min
The smallest value representable by type T.

type(T).max
The largest value representable by type T.

3.7.4 Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

`after`, `alias`, `apply`, `auto`, `byte`, `case`, `copyof`, `default`, `define`, `final`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `var`.

Note

The following identifiers will become keywords in the future and will no longer be usable as names: `at`, `error`, `layout`, `leave`, `super`, `transient`, `this`.

There are also names which will be considered Yul reserved identifiers in the future: `basefee`, `blobbasefee`, `blobhash`, `clz`, `memoryguard`, `mcopy`, `prevrandao`, `tload`, `tstore`.

3.8 Expressions and Control Structures

3.8.1 Control Structures

Most of the control structures known from curly-braces languages are available in Solidity:

There is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from C or JavaScript.

Solidity also supports exception handling in the form of `try`/`catch`-statements, but only for *external function calls* and contract creation calls. Errors can be created using the *revert statement*.

Parentheses can *not* be omitted for conditionals, but curly braces can be omitted around single-statement bodies.

Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

3.8.2 Function Calls

Internal Function Calls

Functions of the current contract can be called directly ("internally"), also recursively, as seen in this nonsensical example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

// This will report a warning
contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

These function calls are translated into simple jumps inside the EVM. This has the effect that the current memory is not cleared, i.e. passing memory references to internally-called functions is very efficient. Only functions of the same contract instance can be called internally.

You should still avoid excessive recursion, as every internal function call uses up at least one stack slot and there are only 1024 slots available.

External Function Calls

Functions can also be called using the `this.g(8);` and `c.g(2);` notation, where `c` is a contract instance and `g` is a function belonging to `c`. Calling the function `g` via either way results in it being called "externally", using a message call and not directly via jumps. Please note that function calls on `this` cannot be used in the constructor, as the actual contract has not been created yet.

Functions of other contracts have to be called externally. For an external call, all function arguments have to be copied to memory.

Note

A function call from one contract to another does not create its own transaction, it is a message call as part of the overall transaction.

When calling functions of other contracts, you can specify the amount of Wei or gas sent with the call with the special options `{value: 10, gas: 10000}`. Note that it is discouraged to specify gas values explicitly, since the gas costs of opcodes can change in the future. Any Wei you send to the contract is added to the total balance of that contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info{value: 10, gas: 800}(); }
}
```

You need to use the modifier `payable` with the `info` function because otherwise, the `value` option would not be available.

Warning

Be careful that `feed.info{value: 10, gas: 800}` only locally sets the `value` and amount of `gas` sent with the function call, and the parentheses at the end perform the actual call. So `feed.info{value: 10, gas: 800}` does not call the function and the `value` and `gas` settings are lost, only `feed.info{value: 10, gas: 800}()` performs the function call.

Warning

Due to the fact that the EVM considers a call to a non-existing contract to always succeed, Solidity uses the

`extcodesize` opcode to check that the contract that is about to be called actually exists (it contains code) and causes an exception if it does not. This check is skipped if the return data will be decoded after the call and thus the ABI decoder will catch the case of a non-existing contract.

This check is not performed in case of *low-level calls* which operate on addresses rather than contract instances.

Warning

Be careful when using high-level calls to *precompiled contracts*, since the compiler considers them non-existing according to the above logic even though they execute code and can return data.

Note

Since the version 0.8.10, the compiler does not check `extcodesize` on high-level external calls if return data is expected, because an empty code will be unable to return data, and the ABI decoder will revert. As a consequence, this allows high-level external calls to precompiled contracts, since they can return data despite having no code associated with their addresses.

Read about *precompiled contracts* and *low-level calls* for more information.

Function calls also cause exceptions if the called contract itself throws an exception or goes out of gas.

Warning

Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

Note

Before Solidity 0.6.2, the recommended way to specify the value and gas was to use `f.value(x).gas(g)()`. This was deprecated in Solidity 0.6.2 and is no longer possible since Solidity 0.7.0.

Function Calls with Named Parameters

Function call arguments can be given by name, in any order, if they are enclosed in `{ }` as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
```

(continues on next page)

(continued from previous page)

```
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }
}
```

Omitted Names in Function Definitions

The names of parameters and return values in the function declaration can be omitted. Those items with omitted names will still be present on the stack, but they are inaccessible by name. An omitted return value name can still return a value to the caller by use of the `return` statement.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

3.8.3 Creating Contracts via `new`

A contract can create other contracts using the `new` keyword. The full code of the contract being created has to be known when the creating contract is compiled so recursive creation-dependencies are not possible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = new D{value: amount}(arg);
```

(continues on next page)

(continued from previous page)

```
        newD.x();
    }
}
```

As seen in the example, it is possible to send Ether while creating an instance of `D` using the `value` option, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

Salted contract creations / create2

When creating a contract, the address of the contract is computed from the address of the creating contract and a counter that is increased with each contract creation.

If you specify the option `salt` (a bytes32 value), then contract creation will use a different mechanism to come up with the address of the new contract:

It will compute the address from the address of the creating contract, the given salt value, the (creation) bytecode of the created contract and the constructor arguments.

In particular, the counter ("nonce") is not used. This allows for more flexibility in creating contracts: You are able to derive the address of the new contract before it is created. Furthermore, you can rely on this address also in case the creating contracts creates other contracts in the meantime.

The main use-case here is contracts that act as judges for off-chain interactions, which only need to be created if there is a dispute.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // This complicated expression just tells you how the address
        // can be pre-computed. It is just there for illustration.
        // You actually only need ``new D{salt: salt}(arg)``.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                abi.encode(arg)
            ))
        )))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

Warning

There are some peculiarities in relation to salted creation. A contract can be re-created at the same address after having been destroyed. Yet, it is possible for that newly created contract to have a different deployed bytecode even though the creation bytecode has been the same (which is a requirement because otherwise the address would change). This is due to the fact that the constructor can query external state that might have changed between the two creations and incorporate that into the deployed bytecode before it is stored.

3.8.4 Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done.

3.8.5 Assignment

Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose number is a constant at compile-time. Those tuples can be used to return multiple values at the same time. These can then either be assigned to newly declared variables or to pre-existing variables (or LValues in general).

Tuples are not proper types in Solidity, they can only be used to form syntactic groupings of expressions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

It is not possible to mix variable declarations and non-declaration assignments, i.e. the following is not valid: `(x, uint y) = (1, 2);`

Note

Prior to version 0.5.0 it was possible to assign to tuples of smaller size, either filling up on the left or on the right side (which ever was empty). This is now disallowed, so both sides have to have the same number of components.

Warning

Be careful when assigning to multiple variables at the same time when reference types are involved, because it could lead to unexpected copying behavior.

Complications for Arrays and Structs

The semantics of assignments are more complicated for non-value types like arrays and structs, including `bytes` and `string`, see *Data location and assignment behavior* for details.

In the example below the call to `g(x)` has no effect on `x` because it creates an independent copy of the storage value in memory. However, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

3.8.6 Scoping and Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the typical "zero-state" of whatever the type is. For example, the default value for a `bool` is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, `bytes` and `string`, the default value is an empty array or string. For the `enum` type, the default value is its first member.

Scoping in Solidity follows the widespread scoping rules of C99 (and many other languages): Variables are visible from the point right after their declaration until the end of the smallest `{ }`-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.

Variables that are parameter-like (function parameters, modifier parameters, catch parameters, ...) are visible inside the code block that follows - the body of the function/modifier for a function and modifier parameter and the catch block for a catch parameter.

Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

As a consequence, the following examples will compile without warnings, since the two variables have the same name but disjoint scopes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

As a special example of the C99 scoping rules, note that in the following, the first assignment to `x` will actually assign the outer and not the inner variable. In any case, you will get a warning about the outer variable being shadowed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// This will report a warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
            uint x;
        }
        return x; // x has value 2
    }
}
```

Warning

Before version 0.5.0 Solidity followed the same scoping rules as JavaScript, that is, a variable declared anywhere within a function would be in scope for the entire function, regardless where it was declared. The following example shows a code snippet that used to compile but leads to an error starting from version 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// This will not compile
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

3.8.7 Checked or Unchecked Arithmetic

An overflow or underflow is the situation where the resulting value of an arithmetic operation, when executed on an unrestricted integer, falls outside the range of the result type.

Prior to Solidity 0.8.0, arithmetic operations would always wrap in case of under- or overflow leading to widespread use of libraries that introduce additional checks.

Since Solidity 0.8.0, all arithmetic operations revert on over- and underflow by default, thus making the use of these libraries unnecessary.

To obtain the previous behavior, an `unchecked` block can be used:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // This subtraction will wrap on underflow.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // This subtraction will revert on underflow.
        return a - b;
    }
}
```

The call to `f(2, 3)` will return `2**256-1`, while `g(2, 3)` will cause a failing assertion.

The `unchecked` block can be used everywhere inside a block, but not as a replacement for a block. It also cannot be nested.

The setting only affects the statements that are syntactically inside the block. Functions called from within an `unchecked` block do not inherit the property.

Note

To avoid ambiguity, you cannot use `_;` inside an `unchecked` block.

The following operators will cause a failing assertion on overflow or underflow and will wrap without an error if used inside an unchecked block:

`++`, `--`, `+`, binary `-`, unary `-`, `*`, `/`, `%`, `**`

`+=`, `-=`, `*=`, `/=`, `%=`

Warning

It is not possible to disable the check for division by zero or modulo by zero using the `unchecked` block.

Note

Bitwise operators do not perform overflow or underflow checks. This is particularly visible when using bitwise shifts (`<<`, `>>`, `<<=`, `>>=`) in place of integer division and multiplication by a power of 2. For example `type(uint256).max << 3` does not revert even though `type(uint256).max * 8` would.

Note

The second statement in `int x = type(int).min; -x;` will result in an overflow because the negative range can hold one more value than the positive range.

Explicit type conversions will always truncate and never cause a failing assertion with the exception of a conversion from an integer to an enum type.

3.8.8 Error handling: Assert, Require, Revert and Exceptions

Solidity uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller.

When exceptions happen in a sub-call, they "bubble up" (i.e., exceptions are rethrown) automatically unless they are caught in a `try/catch` statement. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `staticcall`: they return `false` as their first return value in case of an exception instead of "bubbling up".

Warning

The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

Exceptions can contain error data that is passed back to the caller in the form of *error instances*. The built-in errors `Error(string)` and `Panic(uint256)` are used by special functions, as explained below. `Error` is used for "regular" error conditions while `Panic` is used for errors that should not be present in bug-free code.

Panic via `assert` and Error via `require`

The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met.

The `assert` function creates an error of type `Panic(uint256)`. The same error is created by the compiler in certain situations as listed below.

Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic.

A Panic exception is generated in the following situations. The error code supplied with the error data indicates the kind of panic.

1. 0x00: Used for generic compiler inserted panics.
2. 0x01: If you call `assert` with an argument that evaluates to false.
3. 0x11: If an arithmetic operation results in underflow or overflow outside of an `unchecked { ... }` block.
4. 0x12; If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
5. 0x21: If you convert a value that is too big or negative into an enum type.
6. 0x22: If you access a storage byte array that is incorrectly encoded.
7. 0x31: If you call `.pop()` on an empty array.
8. 0x32: If you access an array, `bytesN` or an array slice at an out-of-bounds or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).

9. 0x41: If you allocate too much memory or create an array that is too large.
10. 0x51: If you call a zero-initialized variable of internal function type.

The `require` function provides three overloads:

1. `require(bool)` which will revert without any data (not even an error selector).
2. `require(bool, string)` which will revert with an `Error(string)`.
3. `require(bool, error)` which will revert with the custom, user supplied error provided as the second argument.

Note

`require` arguments are evaluated unconditionally, so take special care to make sure that they are not expressions with unexpected side-effects. For example, in `require(condition, CustomError(f()));` and `require(condition, f());`, function `f()` will be called regardless of whether the supplied condition is `true` or `false`.

An `Error(string)` exception (or an exception without data) is generated by the compiler in the following situations:

1. Calling `require(x)` where `x` evaluates to `false`.
2. If you use `revert()` or `revert("description")`.
3. If you perform an external function call targeting a contract that contains no code.
4. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
5. If your contract receives Ether via a public getter function.

For the following cases, the error data from the external call (if provided) is forwarded. This means that it can either cause an `Error` or a `Panic` (or whatever else was given):

1. If a `.transfer()` fails.
2. If you call a function via a message call but it does not finish properly (i.e., it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall`, `callcode` or `staticcall` is used. The low level operations never throw exceptions but indicate failures by returning `false`.
3. If you create a contract using the `new` keyword but the contract creation *does not finish properly*.

You can optionally provide a message string or a custom error to `require`, but not to `assert`.

Note

If you do not provide a string or custom error argument to `require`, it will revert with empty error data, not even including the error selector.

The following example shows how you can use `require` to check conditions on inputs and `assert` for internal error checking.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Sharer {
```

(continues on next page)

(continued from previous page)

```
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        (bool success, ) = addr.call{value: msg.value / 2}("");
        require(success);
        // Since require will stop execution and revert if success is false,
        // there should be no way for us to still have half of the Ether.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

Internally, Solidity performs a revert operation (instruction `0xfd`). This causes the EVM to revert all changes made to the state. The reason for reverting is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to keep the atomicity of transactions, the safest action is to revert all changes and make the whole transaction (or at least call) without effect.

In both cases, the caller can react on such failures using `try/catch`, but the changes in the callee will always be reverted.

Note

Panic exceptions used to use the `invalid` opcode before Solidity 0.8.0, which consumed all gas available to the call. Exceptions that use `require` used to consume all gas until before the Metropolis release.

`revert`

A direct revert can be triggered using the `revert` statement and the `revert` function.

The `revert` statement takes a custom error as direct argument without parentheses:

    revert CustomError(arg1, arg2);

For backward-compatibility reasons, there is also the `revert()` function, which uses parentheses and accepts a string:

    revert(); revert("description");

The error data will be passed back to the caller and can be caught there. Using `revert()` causes a revert without any error data while `revert("description")` will create an `Error(string)` error.

Using a custom error instance will usually be much cheaper than a string description, because you can use the name of the error to describe it, which is encoded in only four bytes. A longer description can be supplied via NatSpec which does not incur any costs.

The following example shows how to use an error string and a custom error instance together with `revert` and the equivalent `require`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
```

(continues on next page)

(continued from previous page)

```
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        (bool success, ) = payable(msg.sender).call{value: address(this).balance}("");
        require(success);
    }
}
```

The two ways `if (!condition) revert(...);` and `require(condition, ...);` are equivalent as long as the arguments to `revert` and `require` do not have side-effects, for example if they are just strings.

Note

The `require` function is evaluated just as any other function. This means that all arguments are evaluated before the function itself is executed. In particular, in `require(condition, f())` the function `f` is executed even if `condition` is true.

The provided string is *abi-encoded* as if it were a call to a function `Error(string)`. In the above example, `revert("Not enough Ether provided.");` returns the following hexadecimal as error return data:

```
0x08c379a0                                                         // Function selector␣
↪for Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Data offset
0x000000000000000000000000000000000000000000000000000000000000001a // String length
0x4e6f7420656e6f756768204574686572207072f6696465642e0000000000000 // String data
```

The provided message can be retrieved by the caller using `try/catch` as shown below.

Note

There used to be a keyword called `throw` with the same semantics as `revert()` which was deprecated in version 0.4.13 and removed in version 0.5.0.

try/catch

A failure in an external call can be caught using a try/catch statement, as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
```

(continues on next page)

(continued from previous page)

```
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // This is executed in case of a panic,
            // i.e. a serious error like division by zero
            // or overflow. The error code can be used
            // to determine the kind of error.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used.
            errorCount++;
            return (0, false);
        }
    }
}
```

The `try` keyword has to be followed by an expression representing an external function call or a contract creation (`new ContractName()`). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The `returns` part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract's execution continues inside the first success block. If the end of the success block is reached, execution continues after the `catch` blocks.

Solidity supports different kinds of catch blocks depending on the type of error:

- `catch Error(string memory reason) { ... }`: This catch clause is executed if the error was caused by `revert("reasonString")` or `require(false, "reasonString")` (or an internal error that causes such an exception).
- `catch Panic(uint errorCode) { ... }`: If the error was caused by a panic, i.e. by a failing `assert`, division by zero, invalid array access, arithmetic overflow and others, this catch clause will be run.
- `catch (bytes memory lowLevelData) { ... }`: This clause is executed if the error signature does not match any other clause, if there was an error while decoding the error message, or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.
- `catch { ... }`: If you are not interested in the error data, you can just use `catch { ... }` (even as the only catch clause) instead of the previous clause.

It is planned to support other types of error data in the future. The strings `Error` and `Panic` are currently parsed as is and are not treated as identifiers.

In order to catch all error cases, you have to have at least the clause `catch { ...}` or the clause `catch (bytes`

`memory lowLevelData) { ... }`.

The variables declared in the `returns` and the `catch` clause are only in scope in the block that follows.

Note

If an error happens during the decoding of the return data inside a try/catch-statement, this causes an exception in the currently executing contract and because of that, it is not caught in the catch clause. If there is an error during decoding of `catch Error(string memory reason)` and there is a low-level catch clause, this error is caught there.

Note

If execution reaches a catch-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a catch block or the execution of the try/catch statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).

Note

The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract: The error might have happened deeper down in the call chain and the called contract just forwarded it. Also, it could be due to an out-of-gas situation and not a deliberate error condition: The caller always retains at least 1/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left.

3.9 Contracts

Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables, and functions that can modify these variables. Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables in the calling contract are inaccessible. A contract and its functions need to be called for anything to happen. There is no "cron" concept in Ethereum to call a function at a particular event automatically.

3.9.1 Creating Contracts

Contracts can be created "from outside" via Ethereum transactions or from within Solidity contracts.

IDEs, such as Remix, make the creation process seamless using UI elements.

One way to create contracts programmatically on Ethereum is via the JavaScript API web3.js. It has a function called web3.eth.Contract to facilitate contract creation.

When a contract is created, its *constructor* (a function declared with the `constructor` keyword) is executed once.

A constructor is optional. Only one constructor is allowed, which means overloading is not supported.

After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

Internally, constructor arguments are passed *ABI encoded* after the code of the contract itself, but you do not have to care about this if you use `web3.js`.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


contract OwnedToken {
    // `TokenCreator` is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    constructor(bytes32 name_) {
        // State variables are accessed via their name
        // and not via e.g. `this.owner`. Functions can
        // be accessed directly or through `this.f`,
        // but the latter provides an external view
        // to the function. Especially in the constructor,
        // you should not access functions externally,
        // because the function does not exist yet.
        // See the next section for details.
        owner = msg.sender;

        // We perform an explicit type conversion from `address`
        // to `TokenCreator` and assume that the type of
        // the calling contract is `TokenCreator`, there is
        // no real way to verify that.
        // This does not create a new contract.
        creator = TokenCreator(msg.sender);
        name = name_;
    }

    function changeName(bytes32 newName) public {
        // Only the creator can alter the name.
        // We compare the contract based on its
        // address which can be retrieved by
        // explicit conversion to address.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;

        // We ask the creator contract if the transfer
        // should proceed by using a function of the
        // `TokenCreator` contract defined below. If
```

(continues on next page)

(continued from previous page)

```
        // the call fails (e.g. due to out-of-gas),
        // the execution also fails here.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}


contract TokenCreator {
    function createToken(bytes32 name)
        public
        returns (OwnedToken tokenAddress)
    {
        // Create a new `Token` contract and return its address.
        // From the JavaScript side, the return type
        // of this function is `address`, as this is
        // the closest type available in the ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // Again, the external type of `tokenAddress` is
        // simply `address`.
        tokenAddress.changeName(name);
    }

    // Perform checks to determine if transferring a token to the
    // `OwnedToken` contract should proceed
    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Check an arbitrary condition to see if transfer should proceed
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

3.9.2 Visibility and Getters

State Variable Visibility

public
: Public state variables differ from internal ones only in that the compiler automatically generates *getter functions* for them, which allows other contracts to read their values. When used within the same contract, the external access (e.g. `this.x`) invokes the getter while internal access (e.g. `x`) gets the variable value directly from storage. Setter functions are not generated so other contracts cannot directly modify their values.

internal
: Internal state variables can only be accessed from within the contract they are defined in and in derived contracts. They cannot be accessed externally. This is the default visibility level for state variables.

private
: Private state variables are like internal ones but they are not visible in derived contracts.

Warning

Making something `private` or `internal` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

Function Visibility

Solidity knows two kinds of function calls: external ones that do create an actual EVM message call and internal ones that do not. Furthermore, internal functions can be made inaccessible to derived contracts. This gives rise to four types of visibility for functions.

`external`
: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works).

`public`
: Public functions are part of the contract interface and can be either called internally or via message calls.

`internal`
: Internal functions can only be accessed from within the current contract or contracts deriving from it. They cannot be accessed externally. Since they are not exposed to the outside through the contract's ABI, they can take parameters of internal types like mappings or storage references.

`private`
: Private functions are like internal ones but they are not visible in derived contracts.

Warning

Making something `private` or `internal` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

The visibility specifier is given after the type for state variables and between parameter list and return parameter list for functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

In the following example, `D`, can call `c.getData()` to retrieve the value of `data` in state storage, but is not able to call `f`. Contract `E` is derived from `C` and, thus, can call `compute`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
```

(continues on next page)

(continued from previous page)

```
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}

// This will not compile
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // error: member `f` is not visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member `compute` is not visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // access to internal member (from derived to parent␣
↪contract)
    }
}
```

Getter Functions

The compiler automatically creates getter functions for all **public** state variables. For the contract given below, the compiler will generate a function called `data` that does not take any arguments and returns a `uint`, the value of the state variable `data`. State variables can be initialized when they are declared.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

The getter functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it evaluates to a state variable. If it is accessed externally (i.e. with `this.`), it evaluates to a function.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
```

(continues on next page)

(continued from previous page)

```
        data = 3; // internal access
        return this.data(); // external access
    }
}
```

If you have a `public` state variable of array type, then you can only retrieve single elements of the array via the generated getter function. This mechanism exists to avoid high gas costs when returning an entire array. You can use arguments to specify which individual element to return, for example `myArray(0)`. If you want to return an entire array in one call, then you need to write a function, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {
    // public state variable
    uint[] public myArray;

    // Getter function generated by the compiler
    /*
    function myArray(uint i) public view returns (uint) {
        return myArray[i];
    }
    */

    // function that returns entire array
    function getArray() public view returns (uint[] memory) {
        return myArray;
    }
}
```

Now you can use `getArray()` to retrieve the entire array, instead of `myArray(i)`, which returns a single element per call.

The next example is more complex:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping(uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping(uint => mapping(bool => Data[])) public data;
}
```

It generates a function of the following form. The mapping and arrays (with the exception of byte arrays) in the struct are omitted because there is no good way to select individual struct members or provide a key for the mapping:

```
function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}
```

3.9.3 Function Modifiers

Modifiers can be used to change the behavior of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function.

Modifiers are inheritable properties of contracts and may be overridden by derived contracts, but only if they are marked `virtual`. For details, please see *Modifier Overriding*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;

    // This contract only defines a modifier but does not use
    // it: it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // `_;` in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract priced {
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, owned {
    mapping(address => bool) registeredAddresses;
    uint price;
```

(continues on next page)

(continued from previous page)

```
    constructor(uint initialPrice) { price = initialPrice; }

    // It is important to also provide the
    // `payable` keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    // This contract inherits the `onlyOwner` modifier from
    // the `owned` contract. As a result, calls to `changePrice` will
    // only take effect if they are made by the stored owner.
    function changePrice(uint price_) public onlyOwner {
        price = price_;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// This function is protected by a mutex, which means that
    /// reentrant calls from within `msg.sender.call` cannot call `f` again.
    /// The `return 7` statement assigns 7 to the return value but still
    /// executes the statement `locked = false` in the modifier.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

If you want to access a modifier `m` defined in a contract `C`, you can use `C.m` to reference it without virtual lookup. It is only possible to use modifiers defined in the current contract or its base contracts. Modifiers can also be defined in libraries but their use is limited to functions of the same library.

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

Modifiers cannot implicitly access or change the arguments and return values of functions they modify. Their values can only be passed to them explicitly at the point of invocation.

In function modifiers, it is necessary to specify when you want the function to which the modifier is applied to be run. The placeholder statement (denoted by a single underscore character _) is used to denote where the body of the function being modified should be inserted. Note that the placeholder operator is different from using underscores as leading or trailing characters in variable names, which is a stylistic choice.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the _ in the preceding modifier.

> **Warning**
>
> In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

An explicit return from a modifier with `return;` does not affect the values returned by the function. The modifier can, however, choose not to execute the function body at all and in that case the return variables are set to their *default values* just as if the function had an empty body.

The _ symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body, and the function returns the return value of the final occurrence.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

3.9.4 Transient Storage

Transient storage is another data location besides memory, storage, calldata (and return-data and code) which was introduced alongside its respective opcodes `TSTORE` and `TLOAD` by EIP-1153. This new data location behaves as a key-value store similar to storage with the main difference being that data in transient storage is not permanent, but is scoped to the current transaction only, after which it will be reset to zero. Since the content of transient storage has very limited lifetime and size, it does not need to be stored permanently as a part of state and the associated gas costs are much lower than in case of storage. EVM version `cancun` or newer is required for transient storage to be available.

Transient storage variables cannot be initialized in place, i.e., they cannot be assigned to upon declaration, since the value would be cleared at the end of the creation transaction, rendering the initialization ineffective. Transient variables will be *default value* initialized depending on their underlying type. `constant` and `immutable` variables conflict with transient storage, since their values are either inlined or directly stored in code.

Transient storage variables have completely independent address space from storage, so that the order of transient state variables does not affect the layout of storage state variables and vice-versa. They do need distinct names though because all state variables share the same namespace. It is also important to note that the values in transient storage are packed in the same fashion as those in persistent storage. See *Storage Layout* for more information.

Besides that, transient variables can have visibility as well and `public` ones will have a getter function generated automatically as usual.

Note that, currently, such use of `transient` as a data location is only allowed for *value type* state variable declarations. Reference types, such as arrays, mappings and structs, as well as local or parameter variables are not yet supported.

An expected canonical use case for transient storage is cheaper reentrancy locks, which can be readily implemented with the opcodes as showcased next.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.28;

contract Generosity {
    mapping(address => bool) sentGifts;
    bool transient locked;

    modifier nonReentrant {
        require(!locked, "Reentrancy attempt");
        locked = true;
```

(continues on next page)

(continued from previous page)

```
        _;
        // Unlocks the guard, making the pattern composable.
        // After the function exits, it can be called again, even in the same␣
↪transaction.
        locked = false;
    }

    function claimGift() nonReentrant public {
        require(address(this).balance >= 1 ether);
        require(!sentGifts[msg.sender]);
        (bool success, ) = msg.sender.call{value: 1 ether}("");
        require(success);

        // In a reentrant function, doing this last would open up the vulnerability
        sentGifts[msg.sender] = true;
    }
}
```

Transient storage is private to the contract that owns it, in the same way as persistent storage. Only owning contract frames may access their transient storage, and when they do, all the frames access the same transient store.

Transient storage is part of the EVM state and is subject to the same mutability enforcements as persistent storage. As such, any read access to it is not `pure` and writing access is not `view`.

If the `TSTORE` opcode is called within the context of a `STATICCALL`, it will result in an exception instead of performing the modification. `TLOAD` is allowed within the context of a `STATICCALL`.

When transient storage is used in the context of `DELEGATECALL` or `CALLCODE`, then the owning contract of the transient storage is the contract that issued `DELEGATECALL` or `CALLCODE` instruction (the caller) as with persistent storage. When transient storage is used in the context of `CALL` or `STATICCALL`, then the owning contract of the transient storage is the contract that is the target of the `CALL` or `STATICCALL` instruction (the callee).

Note

In the case of `DELEGATECALL`, since references to transient storage variables are currently not supported, it is not possible to pass those into library calls. In libraries, access to transient storage is only possible using inline assembly.

If a frame reverts, all writes to transient storage that took place between entry to the frame and the return are reverted, including those that took place in inner calls. The caller of an external call may employ a `try ... catch` block to prevent reverts bubbling up from the inner calls.

3.9.5 Composability of Smart Contracts and the Caveats of Transient Storage

Given the caveats mentioned in the specification of EIP-1153, in order to preserve the composability of your smart contract, utmost care is recommended for more advanced use cases of transient storage.

For smart contracts, composability is a very important design principle to achieve self-contained behaviour, such that multiple calls into individual smart contracts can be composed to more complex applications. So far the EVM largely guaranteed composable behaviour, since multiple calls into a smart contract within a complex transaction are virtually indistinguishable from multiple calls to the contract stretched over several transactions. However, transient storage allows a violation of this principle, and incorrect use may lead to complex bugs that only surface when used across several calls.

Let's illustrate the problem with a simple example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.28;

contract MulService {
    uint transient multiplier;
    function setMultiplier(uint mul) external {
        multiplier = mul;
    }

    function multiply(uint value) external view returns (uint) {
        return value * multiplier;
    }
}
```

and a sequence of external calls:

```
setMultiplier(42);
multiply(1);
multiply(2);
```

If the example used memory or storage to store the multiplier, it would be fully composable. It would not matter whether you split the sequence into separate transactions or grouped them in some way. You would always get the same result: after `multiplier` is set to `42`, the subsequent calls would return `42` and `84` respectively. This enables use cases such as batching calls from multiple transactions together to reduce gas costs. Transient storage potentially breaks such use cases since composability can no longer be taken for granted. In the example, if the calls are not executed in the same transaction, then `multiplier` is reset and the next calls to function `multiply` would always return `0`.

As another example, since transient storage is constructed as a relatively cheap key-value store, a smart contract author may be tempted to use transient storage as a replacement for in-memory mappings without keeping track of the modified keys in the mapping and thereby without clearing the mapping at the end of the call. This, however, can easily lead to unexpected behaviour in complex transactions, in which values set by a previous call into the contract within the same transaction remain.

The use of transient storage for reentrancy locks that are cleared at the end of the call frame into the contract, is safe. However, be sure to resist the temptation to save the 100 gas used for resetting the reentrancy lock, since failing to do so, will restrict your contract to only one call within a transaction, preventing its use in complex composed transactions, which have been a cornerstone for complex applications on chain.

It is recommend to generally always clear transient storage completely at the end of a call into your smart contract to avoid these kinds of issues and to simplify the analysis of the behaviour of your contract within complex transactions. Check the Security Considerations section of EIP-1153 for further details.

3.9.6 Constant and Immutable State Variables

State variables can be declared as `constant` or `immutable`. In both cases, the variables cannot be modified after the contract has been constructed. For `constant` variables, the value has to be fixed at compile-time, while for `immutable`, it can still be assigned at construction time.

It is also possible to define `constant` variables at the file level.

Every occurrence of such a variable in the source is replaced by its underlying value and the compiler does not reserve a storage slot for it. It cannot be assigned a slot in transient storage using the `transient` keyword either.

Compared to regular state variables, the gas costs of constant and immutable variables are much lower. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time.

This allows for local optimizations. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, constant values can sometimes be cheaper than immutable values.

Not all types for constants and immutables are implemented at this time. The only supported types are *strings* (only for constants) and *value types*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.21;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals = 18;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint decimals_, address ref) {
        if (decimals_ != 0)
            // Immutables are only immutable when deployed.
            // At construction time they can be assigned to any number of times.
            decimals = decimals_;

        // Assignments to immutables can even access the environment.
        maxBalance = ref.balance;
    }

    function isBalanceTooHigh(address other) public view returns (bool) {
        return other.balance > maxBalance;
    }
}
```

Constant

For `constant` variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. `block.timestamp`, `address(this).balance` or `block.number`) or execution data (`msg.value` or `gasleft()`) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` and `mulmod` are allowed (even though, with the exception of `keccak256`, they do call external contracts).

The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.

Immutable

Variables declared as `immutable` are a bit less restricted than those declared as `constant`: Immutable variables can be assigned a value at construction time. The value can be changed at any time before deployment and then it becomes permanent.

One additional restriction is that immutables can only be assigned to inside expressions for which there is no possibility of being executed after creation. This excludes all modifier definitions and functions other than constructors.

There are no restrictions on reading immutable variables. The read is even allowed to happen before the variable is

written to for the first time because variables in Solidity always have a well-defined initial value. For this reason it is also allowed to never explicitly assign a value to an immutable.

Warning

When accessing immutables at construction time, please keep the *initialization order* in mind. Even if you provide an explicit initializer, some expressions may end up being evaluated before that initializer, especially when they are at a different level in inheritance hierarchy.

Note

Before Solidity 0.8.21 initialization of immutable variables was more restrictive. Such variables had to be initialized exactly once at construction time and could not be read before then.

The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by replacing all references to immutables with the values assigned to them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain. The compiler outputs where these immutables are located in the deployed bytecode in the `immutableReferences` field of the *compiler JSON standard output*.

3.9.7 Custom Storage Layout

A contract can define an arbitrary location for its storage using the `layout` specifier. The contract's state variables, including those inherited from base contracts, start from the specified base slot instead of the default slot zero.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.29;

contract C layout at 0xAAAA + 0x11 {
    uint[3] x; // Occupies slots 0xAABB..0xAABD
}
```

As the above example shows, the specifier uses the `layout at <base-slot-expression>` syntax and is located in the header of a contract definition.

The layout specifier can be placed either before or after the inheritance specifier, and can appear at most once. The `base-slot-expression` must be an *integer literal* expression that can be evaluated at compilation time and yields a value in the range of `uint256`. The use of constants initialized using such expressions and the *built-in function erc7201* is also allowed.

A custom layout cannot make contract's storage "wrap around". If the selected base slot would push the static variables past the end of storage, the compiler will issue an error. Note that the data areas of dynamic arrays and mappings are not affected by this check because their layout is not linear. Regardless of the base slot used, their locations are calculated in a way that always puts them within the range of `uint256` and their sizes are not known at compilation time.

While there are no other limits placed on the base slot, it is recommended to avoid locations that are too close to the end of the address space. Leaving too little space may complicate contract upgrades or cause problems for contracts that store additional values past their allocated space using inline assembly.

The storage layout can only be specified for the topmost contract of an inheritance tree, and affects locations of all the storage variables in all the contracts in that tree. Variables are laid out according to the order of their definitions and the positions of their contracts in the *linearized inheritance hierarchy* and a custom base slot preserves their relative positions, shifting them all by the same amount.

The storage layout cannot be specified for abstract contracts, interfaces and libraries. Also, it is important to note that it does *not* affect transient state variables.

For details about storage layout and the effect of the layout specifier on it see *layout of storage variables*.

Warning

The identifiers `layout` and `at` are not yet reserved as keywords in the language. It is strongly recommended to avoid using them since they will become reserved in a future breaking release.

3.9.8 Functions

Functions can be defined inside and outside of contracts.

Functions outside of a contract, also called "free functions", always have implicit `internal` *visibility*. Their code is included in all contracts that call them, similar to internal library functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory arr) pure returns (uint s) {
    for (uint i = 0; i < arr.length; i++)
        s += arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory arr) public {
        // This calls the free function internally.
        // The compiler will add its code to the contract.
        uint s = sum(arr);
        require(s >= 10);
        found = true;
    }
}
```

Note

Functions defined outside a contract are still always executed in the context of a contract. They still can call other contracts, send them Ether and destroy the contract that called them, among other things. The main difference to functions defined inside a contract is that free functions do not have direct access to the variable `this`, storage variables and functions not in their scope.

Function Parameters and Return Variables

Functions take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output.

Function Parameters

Function parameters are declared the same way as variables, and the name of unused parameters can be omitted.

For example, if you want your contract to accept one kind of external call with two integers, you would use something like the following:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    uint sum;
    function taker(uint a, uint b) public {
        sum = a + b;
    }
}
```

Function parameters can be used as any other local variable and they can also be assigned to.

Return Variables

Function return variables are declared with the same syntax after the `returns` keyword.

For example, suppose you want to return two results: the sum and the product of two integers passed as function parameters, then you use something like:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        sum = a + b;
        product = a * b;
    }
}
```

The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their *default value* and have that value until they are (re-)assigned.

You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or *multiple ones*) directly with the `return` statement:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
```

(continues on next page)

(continued from previous page)

```
        return (a + b, a * b);
    }
}
```

If you use an early `return` to leave a function that has return variables, you must provide return values together with the return statement.

Note

You cannot return some types from non-internal functions. This includes the types listed below and any composite types that recursively contain them:

- mappings,
- internal function types,
- reference types with location set to `storage`,
- multi-dimensional arrays (applies only to *ABI coder v1*),
- structs (applies only to *ABI coder v1*).

This restriction does not apply to library functions because of their different *internal ABI*.

Returning Multiple Values

When a function has multiple return types, the statement `return (v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an *implicit conversion*.

State Mutability

View Functions

Functions can be declared `view` in which case they promise not to modify the state.

Note

If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used when `view` functions are called, which enforces the state to stay unmodified as part of the EVM execution. For library `view` functions `DELEGATECALL` is used, because there is no combined `DELEGATECALL` and `STATICCALL`. This means library `view` functions do not have run-time checks that prevent state modifications. This should not impact security negatively because library code is usually known at compile-time and the static checker performs compile-time checks.

The following statements are considered modifying the state:

1. Writing to state variables (storage and transient storage).
2. *Emitting events*.
3. *Creating other contracts*.
4. Using `selfdestruct`.
5. Sending Ether via calls.
6. Calling any function not marked `view` or `pure`.

7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + block.timestamp;
    }
}
```

Note

`constant` on functions used to be an alias to `view`, but this was dropped in version 0.5.0.

Note

Getter methods are automatically marked `view`.

Note

Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `view` functions. This enabled state modifications in `view` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `view` functions, modifications to the state are prevented on the level of the EVM.

Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state. In particular, it should be possible to evaluate a `pure` function at compile-time given only its inputs and `msg.data`, but without any knowledge of the current blockchain state. This means that reading from `immutable` variables can be a non-pure operation.

Note

If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used, which does not guarantee that the state is not read, but at least that it is not modified.

In addition to the list of state modifying statements explained above, the following are considered reading from the state:

1. Reading from state variables (storage and transient storage).
2. Accessing `address(this).balance` or `<address>.balance`.
3. Accessing any of the members of `block`, `tx`, `msg` (with the exception of `msg.sig` and `msg.data`).
4. Calling any function not marked `pure`.
5. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

Pure functions are able to use the `revert()` and `require()` functions to revert potential state changes when an *error occurs*.

Reverting a state change is not considered a "state modification", as only changes to the state made previously in code that did not have the `view` or `pure` restriction are reverted and that code has the option to catch the `revert` and not pass it on.

This behavior is also in line with the `STATICCALL` opcode.

Warning

It is not possible to prevent functions from reading the state at the level of the EVM, it is only possible to prevent them from writing to the state (i.e. only `view` can be enforced at the EVM level, `pure` can not).

Note

Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `pure` functions. This enabled state modifications in `pure` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `pure` functions, modifications to the state are prevented on the level of the EVM.

Note

Prior to version 0.4.17 the compiler did not enforce that `pure` is not reading the state. It is a compile-time type check, which can be circumvented by doing invalid explicit conversions between contract types, because the compiler can verify that the type of the contract does not do state-changing operations, but it cannot check that the contract that will be called at runtime is actually of that type.

Special Functions

Receive Ether Function

A contract can have at most one `receive` function, declared using `receive() external payable { ... }` (without the `function` keyword). This function cannot have arguments, cannot return anything and must have `external` visibility and `payable` state mutability. It can be virtual, can override and can have modifiers.

The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers (e.g. via `.send()` or `.transfer()`). If no such function exists, but a payable *fallback function* exists, the fallback function will be called on a plain Ether transfer. If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through a transaction that does not represent a payable function call and throws an exception.

In the worst case, the `receive` function can only rely on 2300 gas being available (for example when `send` or `transfer` is used), leaving little room to perform other operations except basic logging. The following operations will consume

more gas than the 2300 gas stipend:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Warning

`send()` and `transfer()` are deprecated and scheduled for removal. See the section on *send* and *transfer* for more information.

Warning

When Ether is sent directly to a contract (without a function call, i.e. sender uses `send` or `transfer`) but the receiving contract does not define a receive Ether function or a payable fallback function, an exception will be thrown, sending back the Ether (this was different before Solidity v0.4.0). If you want your contract to receive Ether, you have to implement a receive Ether function (using payable fallback functions for receiving Ether is not recommended, since the fallback is invoked and would not fail for interface confusions on the part of the sender).

Warning

A contract without a receive Ether function can receive Ether as a recipient of a *coinbase transaction* (aka *miner block reward*) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This is a design choice of the EVM and Solidity cannot work around it.

It also means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

Below you can see an example of a Sink contract that uses function `receive`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fallback Function

A contract can have at most one `fallback` function, declared using either `fallback () external [payable]` or `fallback (bytes calldata input) external [payable] returns (bytes memory output)` (both without the `function` keyword). This function must have `external` visibility. A fallback function can be virtual, can override and can have modifiers.

The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no *receive Ether function*. The fallback function always receives data, but in order to also receive Ether it must be marked `payable`.

If the version with parameters is used, `input` will contain the full data sent to the contract (equal to `msg.data`) and can return data in `output`. The returned data will not be ABI-encoded. Instead it will be returned without modifications (not even padding).

In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available (see *receive Ether function* for a brief description of the implications of this).

Like any function, the fallback function can execute complex operations as long as there is enough gas passed on to it.

Warning

A `payable` fallback function is also executed for plain Ether transfers, if no *receive Ether function* is present. It is recommended to always define a receive Ether function as well, if you define a payable fallback function to distinguish Ether transfers from interface confusions.

Note

If you want to decode the input data, you can check the first four bytes for the function selector and then you can use `abi.decode` together with the array slice syntax to decode ABI-encoded data: `(c, d) = abi.decode(input[4:], (uint256, uint256));` Note that this should only be used as a last resort and proper functions should be used instead.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the `payable`
    // modifier.
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // This function is called for all messages sent to
    // this contract, except plain Ether transfers
    // (there is no other function except the receive function).
    // Any call with non-empty calldata to this contract will execute
```

(continues on next page)

(continued from previous page)

```
    // the fallback function (even if Ether is sent along with the call).
    fallback() external payable { x = 1; y = msg.value; }

    // This function is called for plain Ether transfers, i.e.
    // for every call with empty calldata.
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1.

        // address(test) will not allow to call ``send`` directly, since ``test`` has no␣
↪payable
        // fallback function.
        // It has to be converted to the ``address payable`` type to even allow calling␣
↪``send`` on it.
        address payable testPayable = payable(address(test));

        // If someone sends Ether to that contract,
        // the transfer will fail, i.e. this returns false here.
        // This will report a warning (deprecation)
        return testPayable.send(2 ether);
    }

    function callTestPayable(TestPayable test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 0.
        (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 1.

        // If someone sends Ether to that contract, the receive function in TestPayable␣
↪will be called.
        // Since that function writes to storage, it takes more gas than is available␣
↪with a
        // simple ``send`` or ``transfer``. Because of that, we have to use a low-level␣
↪call.
        (success,) = address(test).call{value: 2 ether}("");
        require(success);
        // results in test.x becoming == 2 and test.y becoming 2 ether.

        return true;
    }
}
```

Function Overloading

A contract can have multiple functions of the same name but with different parameter types. This process is called "overloading" and also applies to inherited functions. The following example shows overloading of the function `f` in the scope of contract `A`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint value) public pure returns (uint out) {
        out = value;
    }

    function f(uint value, bool really) public pure returns (uint out) {
        if (really)
            out = value;
    }
}
```

Overloaded functions are also present in the external interface. It is an error if two externally visible functions differ by their Solidity types but not by their external types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

// This will not compile
contract A {
    function f(B value) public pure returns (B out) {
        out = value;
    }

    function f(address value) public pure returns (address out) {
        out = value;
    }
}

contract B {
}
```

Both `f` function overloads above end up accepting the address type for the ABI although they are considered different inside Solidity.

Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

> **Note**
>
> Return parameters are not taken into account for overload resolution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint8 val) public pure returns (uint8 out) {
        out = val;
    }

    function f(uint256 val) public pure returns (uint256 out) {
        out = val;
    }
}
```

Calling `f(50)` would create a type error since `50` can be implicitly converted both to `uint8` and `uint256` types. On another hand `f(256)` would resolve to `f(uint256)` overload as `256` cannot be implicitly converted to `uint8`.

3.9.9 Events

Solidity events give an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Events can be defined at file level or as inheritable members of contracts (including interfaces and libraries). When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract that emitted them, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of now, but this might change in the future). The Log and its event data are not accessible from within contracts (not even from the contract that created them).

It is possible to request a Merkle proof for logs, so if an external entity supplies a contract with such a proof, it can check that the log actually exists inside the blockchain. You have to supply block headers because the contract can only see the last 256 block hashes.

You can add the attribute `indexed` to up to three parameters which adds them to a special data structure known as *"topics"* instead of the data part of the log. A topic can only hold a single word (32 bytes) so if you use a *reference type* for an indexed argument, the Keccak-256 hash of the value is stored as a topic instead.

All parameters without the `indexed` attribute are *ABI-encoded* into the data part of the log.

Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

For example, the code below uses the web3.js `subscribe("logs")` method to filter logs that match a topic with a certain address value:

```
var options = {
    fromBlock: 0,
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000", null,␣
↪null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
```

(continues on next page)

(continued from previous page)

```
    .on("changed", function (log) {
});
```

The hash of the signature of the event is one of the topics, except if you declared the event with the `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name, you can only filter by the contract address. The advantage of anonymous events is that they are cheaper to deploy and call. It also allows you to declare four indexed arguments rather than three.

Note

Since the transaction log only stores the event data and not the type, you have to know the type of the event, including which parameter is indexed and if the event is anonymous in order to correctly interpret the data. In particular, it is possible to "fake" the signature of another event using an anonymous event.

Members of Events

- `event.selector`: For non-anonymous events, this is a `bytes32` value containing the `keccak256` hash of the event signature, as used in the default topic.

Example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed from,
        bytes32 indexed id,
        uint value
    );

    function deposit(bytes32 id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, id, msg.value);
    }
}
```

The use in the JavaScript API is as follows:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var depositEvent = clientReceipt.Deposit();

// watch for changes
depositEvent.watch(function(error, result){
```

(continues on next page)

(continued from previous page)

```
    // result contains non-indexed arguments and topics
    // given to the `Deposit` call.
    if (!error)
        console.log(result);
});


// Or pass a callback to start watching immediately
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

The output of the above looks like the following (trimmed):

```
{
   "returnValues": {
       "from": "0x1111...FFFFCCCC",
       "id": "0x50...sd5adb20",
       "value": "0x420042"
   },
   "raw": {
       "data": "0x7f...91385",
       "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
   }
}
```

Additional Resources for Understanding Events

- JavaScript documentation
- Example usage of events
- How to access them in js

3.9.10 Custom Errors

Errors in Solidity provide a convenient and gas-efficient way to explain to the user why an operation failed. They can be defined inside and outside of contracts (including interfaces and libraries).

They have to be used together with the *revert statement* or the *require function*. In the case of `revert` statements, or `require` calls where the condition is evaluated to be false, all changes in the current call are reverted, and the error data passed back to the caller.

The example below shows custom error usage with the `revert` statement in function `transferWithRevertError`, as well as the newer approach with `require` in function `transferWithRequireError`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.27;

/// Insufficient balance for transfer. Needed `required` but only
/// `available` available.
/// @param available balance available.
/// @param required requested amount to transfer.
```

(continues on next page)

(continued from previous page)

```
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
    mapping(address => uint) balance;
    function transferWithRevertError(address to, uint256 amount) public {
        if (amount > balance[msg.sender])
            revert InsufficientBalance({
                available: balance[msg.sender],
                required: amount
            });
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    function transferWithRequireError(address to, uint256 amount) public {
        require(amount <= balance[msg.sender], InsufficientBalance(balance[msg.sender],␣
↪amount));
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    // ...
}
```

Another important detail to mention when it comes to using `require` with custom errors, is that memory allocation for the error-based revert reason will only happen in the reverting case, which, along with optimization of constants and string literals makes this about as gas-efficient as the `if (!condition) revert CustomError(args)` pattern.

Errors cannot be overloaded or overridden but are inherited. The same error can be defined in multiple places as long as the scopes are distinct. Instances of errors can only be created using `revert` statements, or as the second argument to `require` functions.

The error creates data that is then passed to the caller with the revert operation to either return to the off-chain component or catch it in a *try/catch statement*. Note that an error can only be caught when coming from an external call, reverts happening in internal calls or inside the same function cannot be caught.

If you do not provide any parameters, the error only needs four bytes of data and you can use *NatSpec* as above to further explain the reasons behind the error, which is not stored on chain. This makes this a very cheap and convenient error-reporting feature at the same time.

More specifically, an error instance is ABI-encoded in the same way as a function call to a function of the same name and types would be and then used as the return data in the `revert` opcode. This means that the data consists of a 4-byte selector followed by *ABI-encoded* data. The selector consists of the first four bytes of the keccak256-hash of the signature of the error type.

Note

It is possible for a contract to revert with different errors of the same name or even with errors defined in different places that are indistinguishable by the caller. For the outside, i.e. the ABI, only the name of the error is relevant, not the contract or file where it is defined.

The statement `require(condition, "description");` would be equivalent to `if (!condition) revert Error("description")` if you could define `error Error(string)`. Note, however, that `Error` is a built-in type and cannot be defined in user-supplied code.

Similarly, a failing `assert` or similar conditions will revert with an error of the built-in type `Panic(uint256)`.

> **Note**
>
> Error data should only be used to give an indication of failure, but not as a means for control-flow. The reason is that the revert data of inner calls is propagated back through the chain of external calls by default. This means that an inner call can "forge" revert data that looks like it could have come from the contract that called it.

Members of Errors

- `error.selector`: A `bytes4` value containing the error selector.

3.9.11 Inheritance

Solidity supports multiple inheritance including polymorphism.

Polymorphism means that a function call (internal and external) always executes the function of the same name (and parameter types) in the most derived contract in the inheritance hierarchy. This has to be explicitly enabled on each function in the hierarchy using the `virtual` and `override` keywords. See *Function Overriding* for more details.

It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using `ContractName.functionName()` or using `super.functionName()` if you want to call the function one level higher up in the flattened inheritance hierarchy (see below).

When a contract inherits from other contracts, only a single contract is created on the blockchain, and the code from all the base contracts is compiled into the created contract. This means that all internal calls to functions of base contracts also just use internal function calls (`super.f(..)` will use JUMP and not a message call).

State variable shadowing is considered as an error. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

The general inheritance system is very similar to Python's, especially concerning multiple inheritance, but there are also some *differences*.

Details are given in the following example.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Owned {
    address payable owner;
    constructor() { owner = payable(msg.sender); }
}

// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract Emittable is Owned {
    event Emitted();

    // The keyword `virtual` means that the function can change
    // its behavior in derived classes ("overriding").
    function emitEvent() virtual public {
        if (msg.sender == owner)
            emit Emitted();
    }
}
```

(continues on next page)

(continued from previous page)

```
// These abstract contracts are only provided to make the
// interface known to the compiler. Note the function
// without body. If a contract does not implement all
// functions it can only be used as an interface.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}

abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}

// Multiple inheritance is possible. Note that `Owned` is
// also a base class of `Emittable`, yet there is only a single
// instance of `Owned` (as for virtual inheritance in C++).
contract Named is Owned, Emittable {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs. If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    // If you want the function to override, you need to use the
    // `override` keyword. You need to specify the `virtual` keyword again
    // if you want this function to be overridden again.
    function emitEvent() public virtual override {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific
            // overridden function.
            Emittable.emitEvent();
        }
    }
}


// If a constructor takes an argument, it needs to be
// provided in the header or modifier-invocation-style at
// the constructor of the derived contract (see below).
contract PriceFeed is Owned, Emittable, Named("GoldFeed") {
    uint info;

    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }
```

(continues on next page)

(continued from previous page)

```
    // Here, we only specify `override` and not `virtual`.
    // This means that contracts deriving from `PriceFeed`
    // cannot change the behavior of `emitEvent` anymore.
    function emitEvent() public override(Emittable, Named) { Named.emitEvent(); }
    function get() public view returns(uint r) { return info; }
}
```

Note that above, we call `Emittable.emitEvent()` to "forward" the emit event request. The way this is done is problematic, as seen in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Owned {
    address payable owner;
    constructor() { owner = payable(msg.sender); }
}

contract Emittable is Owned {
    event Emitted();

    function emitEvent() virtual public {
        if (msg.sender == owner) {
            emit Emitted();
        }
    }
}

contract Base1 is Emittable {
    event Base1Emitted();
    function emitEvent() public virtual override {
        /* Here, we emit an event to simulate some Base1 logic */
        emit Base1Emitted();
        Emittable.emitEvent();
    }
}

contract Base2 is Emittable {
    event Base2Emitted();
    function emitEvent() public virtual override {
        /* Here, we emit an event to simulate some Base2 logic */
        emit Base2Emitted();
        Emittable.emitEvent();
    }
}

contract Final is Base1, Base2 {
    event FinalEmitted();
    function emitEvent() public override(Base1, Base2) {
        /* Here, we emit an event to simulate some Final logic */
        emit FinalEmitted();
```

(continues on next page)

(continued from previous page)

```
        Base2.emitEvent();
    }
}
```

A call to `Final.emitEvent()` will call `Base2.emitEvent` because we specify it explicitly in the final override, but this function will bypass `Base1.emitEvent`, resulting in the following sequence of events: `FinalEmitted -> Base2Emitted -> Emitted`, instead of the expected sequence: `FinalEmitted -> Base2Emitted -> Base1Emitted -> Emitted`. The way around this is to use `super`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Owned {
    address payable owner;
    constructor() { owner = payable(msg.sender); }
}

contract Emittable is Owned {
    event Emitted();

    function emitEvent() virtual public {
        if (msg.sender == owner) {
            emit Emitted();
        }
    }
}

contract Base1 is Emittable {
    event Base1Emitted();
    function emitEvent() public virtual override {
        /* Here, we emit an event to simulate some Base1 logic */
        emit Base1Emitted();
        super.emitEvent();
    }
}


contract Base2 is Emittable {
    event Base2Emitted();
    function emitEvent() public virtual override {
        /* Here, we emit an event to simulate some Base2 logic */
        emit Base2Emitted();
        super.emitEvent();
    }
}

contract Final is Base1, Base2 {
    event FinalEmitted();
    function emitEvent() public override(Base1, Base2) {
        /* Here, we emit an event to simulate some Final logic */
        emit FinalEmitted();
        super.emitEvent();
```

(continues on next page)

(continued from previous page)

```
    }
}
```

If `Final` calls a function of `super`, it does not simply call this function on one of its base contracts. Rather, it calls this function on the next base contract in the final inheritance graph, so it will call `Base1.emitEvent()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base2, Base1, Emittable, Owned). The actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

Function Overriding

Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header. The overriding function may only change the visibility of the overridden function from `external` to `public`. The mutability may be changed to a more strict one following the order: `nonpayable` can be overridden by `view` and `pure`. `view` can be overridden by `pure`. `payable` is an exception and cannot be changed to any other mutability.

The following example demonstrates changing mutability and visibility:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

For multiple inheritance, the most derived base contracts that define the same function must be specified explicitly after the `override` keyword. In other words, you have to specify all base contracts that define the same function and have not yet been overridden by another base contract (on some path through the inheritance graph). Additionally, if a contract inherits the same function from multiple (unrelated) bases, it has to explicitly override it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
```

(continues on next page)

(continued from previous page)

```
{
    // Derives from multiple bases defining foo(), so we must explicitly
    // override it
    function foo() public override(Base1, Base2) {}
}
```

An explicit override specifier is not required if the function is defined in a common base contract or if there is a unique function in a common base contract that already overrides all other functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// No explicit override required
contract D is B, C {}
```

More formally, it is not required to override a function (directly or indirectly) inherited from multiple bases if there is a base contract that is part of all override paths for the signature, and (1) that base implements the function and no paths from the current contract to the base mentions a function with that signature or (2) that base does not implement the function and there is at most one mention of the function in all paths from the current contract to that base.

In this sense, an override path for a signature is a path through the inheritance graph that starts at the contract under consideration and ends at a contract mentioning a function with that signature that does not override.

If you do not mark a function that overrides as `virtual`, derived contracts can no longer change the behavior of that function.

Note

Functions with the `private` visibility cannot be `virtual`.

Note

Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered `virtual`.

Note

Starting from Solidity 0.8.8, the `override` keyword is not required when overriding an interface function, except for the case where the function is defined in multiple bases.

Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;
```

(continues on next page)

(continued from previous page)

```
contract A
{
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

Note

While public state variables can override external functions, they themselves cannot be overridden.

Modifier Overriding (deprecated)

Function modifiers can override each other. This works in the same way as *function overriding* (except that there is no overloading for modifiers). The `virtual` keyword must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base
{
    // This will report a warning (deprecation)
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

In case of multiple inheritance, all direct base contracts must be specified explicitly:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    // This will report a warning (deprecation)
    modifier foo() virtual {_;}
}

contract Base2
{
    // This will report a warning (deprecation)
    modifier foo() virtual {_;}
}
```

(continues on next page)

(continued from previous page)

```
contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

Warning

`virtual` modifiers are deprecated and scheduled for removal.

Constructors

A constructor is an optional function declared with the `constructor` keyword which is executed upon contract creation, and where you can run contract initialization code.

Before the constructor code is executed, state variables are initialised to their specified value if you initialise them inline, or their *default value* if you do not.

After the constructor has run, the final code of the contract is deployed to the blockchain. The deployment of the code costs additional gas linear to the length of the code. This code includes all functions that are part of the public interface and all functions that are reachable from there through function calls. It does not include the constructor code or internal functions that are only called from the constructor.

If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() {}`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint a_) {
        a = a_;
    }
}

contract B is A(1) {
    constructor() {}
}
```

You can use internal parameters in a constructor (for example storage pointers). In this case, the contract has to be marked *abstract*, because these parameters cannot be assigned valid values from outside but only through the constructors of derived contracts.

Warning

Prior to version 0.4.22, constructors were defined as functions with the same name as the contract. This syntax was deprecated and is not allowed anymore in version 0.5.0.

Warning

Prior to version 0.7.0, you had to specify the visibility of constructors as either `internal` or `public`.

Arguments for Base Constructors

The constructors of all the base contracts will be called following the linearization rules explained below. If the base constructors have arguments, derived contracts need to specify all of them. This can be done in two ways:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint x_) { x = x_; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() {}
}

// or through a "modifier" of the derived constructor...
contract Derived2 is Base {
    constructor(uint y) Base(y * y) {}
}

// or declare abstract...
abstract contract Derived3 is Base {
}

// and have the next concrete derived contract initialize it.
contract DerivedFromDerived is Derived3 {
    constructor() Base(10 + 10) {}
}
```

One way is directly in the inheritance list (`is Base(7)`). The other is in the way a modifier is invoked as part of the derived constructor (`Base(y * y)`). The first way to do it is more convenient if the constructor argument is a constant and defines the behavior of the contract or describes it. The second way has to be used if the constructor arguments of the base depend on those of the derived contract. Arguments have to be given either in the inheritance list or in modifier-style in the derived constructor. Specifying arguments in both places is an error.

If a derived contract does not specify the arguments to all of its base contracts' constructors, it must be declared abstract. In that case, when another contract derives from it, that other contract's inheritance list or constructor must provide the necessary parameters for all base classes that haven't had their parameters specified (otherwise, that other contract must be declared abstract as well). For example, in the above code snippet, see `Derived3` and `DerivedFromDerived`.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses C3 Linearization to force a specific order in the directed acyclic graph (DAG) of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important: You have to list the direct base contracts in the order from "most base-like" to "most derived". Note that this order is the reverse of the one used in Python.

Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

In the following code, Solidity will give the error "Linearization of inheritance graph impossible".

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract X {}
contract A is X {}
// This will not compile
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.

Due to the fact that you have to explicitly override a function that is inherited from multiple bases without a unique override, C3 linearization is not too important in practice.

One area where inheritance linearization is especially important and perhaps not as clear is when there are multiple constructors in the inheritance hierarchy. The constructors will always be executed in the linearized order, regardless of the order in which their arguments are provided in the inheriting contract's constructor. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}

// Constructors are executed in the following order:
//  1 - Base1
//  2 - Base2
//  3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}

// Constructors are executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// Constructors are still executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived3
```

(continues on next page)

(continued from previous page)

```
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}
```

Inheriting Different Kinds of Members of the Same Name

The only situations where, due to inheritance, a contract may contain multiple definitions sharing the same name are:

- Overloading of functions.
- Overriding of virtual functions.
- Overriding of external virtual functions by state variable getters.
- Overriding of virtual modifiers.
- Overloading of events.

3.9.12 Abstract Contracts

Contracts must be marked as abstract when at least one of their functions is not implemented or when they do not provide arguments for all of their base contract constructors. Even if this is not the case, a contract may still be marked abstract, such as when you do not intend for the contract to be created directly. Abstract contracts are similar to *Interfaces* but an interface is more limited in what it can declare.

An abstract contract is declared using the `abstract` keyword as shown in the following example. Note that this contract needs to be defined as abstract, because the function `utterance()` is declared, but no implementation was provided (no implementation body `{ }` was given).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}
```

Such abstract contracts can not be instantiated directly. This is also true, if an abstract contract itself does implement all defined functions. The usage of an abstract contract as a base class is shown in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public pure virtual returns (bytes32);
}

contract Cat is Feline {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it needs to be marked as abstract as well.

Note that a function without implementation is different from a *Function Type* even though their syntax looks very similar.

Example of function without implementation (a function declaration):

```
function foo(address) external returns (address);
```

Example of a declaration of a variable whose type is a function type:

```
function(address) external returns (address) foo;
```

Abstract contracts decouple the definition of a contract from its implementation providing better extensibility and self-documentation and facilitating patterns like the Template method and removing code duplication. Abstract contracts are useful in the same way that defining methods in an interface is useful. It is a way for the designer of the abstract contract to say "any child of mine must implement this method".

Note

Abstract contracts cannot override an implemented virtual function with an unimplemented one.

3.9.13 Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions:

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external in the interface, even if they are public in the contract.
- They cannot declare a constructor.
- They cannot declare state variables.
- They cannot declare modifiers.

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Contracts can inherit interfaces as they would inherit other contracts.

All functions declared in interfaces are implicitly `virtual` and any functions that override them do not need the `override` keyword. This does not automatically mean that an overriding function can be overridden again - this is only possible if the overriding function is marked `virtual`.

Interfaces can inherit from other interfaces. This has the same rules as normal inheritance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;
```

(continues on next page)

(continued from previous page)

```
interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Must redefine test in order to assert that the parent
    // meanings are compatible.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Types defined inside interfaces and other contract-like structures can be accessed from other contracts: `Token.TokenType` or `Token.Coin`.

Warning

Interfaces have supported `enum` types since *Solidity version 0.5.0*, make sure the pragma version specifies this version as a minimum.

3.9.14 Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the `DELEGATECALL` (`CALLCODE` until Homestead) feature of the EVM. This means that if library functions are called, their code is executed in the context of the calling contract, i.e. `this` points to the calling contract, and especially the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise). Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless. In particular, it is not possible to destroy a library.

Note

Until version 0.4.20, it was possible to destroy libraries by circumventing Solidity's type system. Starting from that version, libraries contain a *mechanism* that disallows state-modifying functions to be called directly (i.e. without `DELEGATECALL`).

Libraries can be seen as implicit base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts (using qualified access like `L.f()`). Of course, calls to internal functions use the internal calling convention, which means that all internal types can be passed and types *stored in memory* will be passed by reference and not copied. To realize this in the EVM, the code of internal library functions that are called from a contract and all functions called from therein will at compile time be included in the calling contract, and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

Note

The inheritance analogy breaks down when it comes to public functions. Calling a public library function with `L.f()` results in an external call (`DELEGATECALL` to be precise). In contrast, `A.f()` is an internal call when `A` is a

base contract of the current contract.

The following example illustrates how to use libraries (but using a manual method, be sure to check out *using for* for a more advanced example to implement a set).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;


// We define a new struct datatype that will be used to
// hold its data in the calling contract.
struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // Note that the first parameter is of type "storage
    // reference" and thus only its storage address and not
    // its contents is passed as part of the call.  This is a
    // special feature of library functions.  It is idiomatic
    // to call the first parameter `self`, if the function can
    // be seen as a method of that object.
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // already there
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}
```

(continues on next page)

(continued from previous page)

```
contract C {
    Data knownValues;

    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}
```

Of course, you do not have to follow this way to use libraries: they can also be used without defining struct data types. Functions also work without any storage reference parameters, and they can have multiple storage reference parameters and in any position.

The calls to `Set.contains`, `Set.insert` and `Set.remove` are all compiled as calls (`DELEGATECALL`) to an external contract/library. If you use libraries, be aware that an actual external function call is performed. `msg.sender`, `msg.value` and `this` will retain their values in this call, though (prior to Homestead, because of the use of `CALLCODE`, `msg.sender` and `msg.value` changed, though).

The following example shows how to use *types stored in memory* and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

struct bigint {
    uint[] limbs;
}

library BigInt {
    function fromUint(uint x) internal pure returns (bigint memory r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint memory a, bigint memory b) internal pure returns (bigint memory␣
↪r) {
        r.limbs = new uint[](max(a.limbs.length, b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint limbA = limb(a, i);
            uint limbB = limb(b, i);
            unchecked {
                r.limbs[i] = limbA + limbB + carry;

                if (limbA + limbB < limbA || (limbA + limbB == type(uint).max && carry >␣
↪0))
                    carry = 1;
                else
                    carry = 0;
            }
```

(continues on next page)

(continued from previous page)

```
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint memory a, uint index) internal pure returns (uint) {
        return index < a.limbs.length ? a.limbs[index] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(type(uint).max);
        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

It is possible to obtain the address of a library by converting the library type to the `address` type, i.e. using `address(LibraryName)`.

As the compiler does not know the address where the library will be deployed, the compiled hex code will contain placeholders of the form `__$30bbc0abd4d6364515865950d3e0d10953$__` (format was different <v0.5.0). The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name, which would be for example `libraries/bigint.sol:BigInt` if the library was stored in a file called `bigint.sol` in a `libraries/` directory. Such bytecode is incomplete and should not be deployed. Placeholders need to be replaced with actual addresses. You can do that by either passing them to the compiler when the library is being compiled or by using the linker to update an already compiled binary. See *Library Linking* for information on how to use the commandline compiler for linking.

In comparison to contracts, libraries are restricted in the following ways:

- they cannot have state variables
- they cannot inherit nor be inherited
- they cannot receive Ether
- they cannot be destroyed

(These might be lifted at a later point.)

Function Signatures and Selectors in Libraries

While external calls to public or external library functions are possible, the calling convention for such calls is considered to be internal to Solidity and not the same as specified for the regular *contract ABI*. External library functions support more argument types than external contract functions, for example recursive structs and storage pointers. For that reason, the function signatures used to compute the 4-byte selector are computed following an internal naming schema and arguments of types not supported in the contract ABI use an internal encoding.

The following identifiers are used for the types in the signatures:

- Value types, non-storage `string` and non-storage `bytes` use the same identifiers as in the contract ABI.
- Non-storage array types follow the same convention as in the contract ABI, i.e. `<type>[]` for dynamic arrays and `<type>[M]` for fixed-size arrays of `M` elements.
- Non-storage structs are referred to by their fully qualified name, i.e. `C.S` for `contract C { struct S { ... } }`.
- Storage pointer mappings use `mapping(<keyType> => <valueType>) storage` where `<keyType>` and `<valueType>` are the identifiers for the key and value types of the mapping, respectively.
- Other storage pointer types use the type identifier of their corresponding non-storage type, but append a single space followed by `storage` to it.

The argument encoding is the same as for the regular contract ABI, except for storage pointers, which are encoded as a `uint256` value referring to the storage slot to which they point.

Similarly to the contract ABI, the selector consists of the first four bytes of the Keccak256-hash of the signature. Its value can be obtained from Solidity using the `.selector` member as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Call Protection For Libraries

As mentioned in the introduction, if a library's code is executed using a `CALL` instead of a `DELEGATECALL` or `CALLCODE`, it will revert unless a `view` or `pure` function is called.

The EVM does not provide a direct way for a contract to detect whether it was called using `CALL` or not, but a contract can use the `ADDRESS` opcode to find out "where" it is currently running. The generated code compares this address to the address used at construction time to determine the mode of calling.

More specifically, the runtime code of a library always starts with a push instruction, which is a zero of 20 bytes at compilation time. When the deploy code runs, this constant is replaced in memory by the current address and this modified code is stored in the contract. At runtime, this causes the deploy time address to be the first constant to be pushed onto the stack and the dispatcher code compares the current address against this constant for any non-view and non-pure function.

This means that the actual code stored on chain for a library is different from the code reported by the compiler as `deployedBytecode`.

3.9.15 Using For

The directive `using A for B` can be used to attach functions (`A`) as operators to user-defined value types or as member functions to any type (`B`). The member functions receive the object they are called on as their first parameter (like the `self` variable in Python). The operator functions receive operands as parameters.

It is valid either at file level or inside a contract, at contract level.

The first part, `A`, can be one of:

- A list of functions, optionally with an operator name assigned (e.g. `using {f, g as +, h, L.t} for uint`). If no operator is specified, the function can be either a library function or a free function and is attached to the type as a member function. Otherwise it must be a free function and it becomes the definition of that operator on the type.
- The name of a library (e.g. `using L for uint`) - all non-private functions of the library are attached to the type as member functions

At file level, the second part, `B`, has to be an explicit type (without data location specifier). Inside contracts, you can also use `*` in place of the type (e.g. `using L for *;`), which has the effect that all functions of the library `L` are attached to *all* types.

If you specify a library, *all* non-private functions in the library get attached, even those where the type of the first parameter does not match the type of the object. The type is checked at the point the function is called and function overload resolution is performed.

If you use a list of functions (e.g. `using {f, g, h, L.t} for uint`), then the type (`uint`) has to be implicitly convertible to the first parameter of each of these functions. This check is performed even if none of these functions are called. Note that private library functions can only be specified when `using for` is inside a library.

If you define an operator (e.g. `using {f as +} for T`), then the type (`T`) must be a *user-defined value type* and the definition must be a `pure` function. Operator definitions must be global. The following operators can be defined this way:

Category	Operator	Possible signatures
Bitwise	`&`	`function (T, T) pure returns (T)`
	`|`	`function (T, T) pure returns (T)`
	`^`	`function (T, T) pure returns (T)`
	`~`	`function (T) pure returns (T)`
Arithmetic	`+`	`function (T, T) pure returns (T)`
	`-`	`function (T, T) pure returns (T)`
		`function (T) pure returns (T)`
	`*`	`function (T, T) pure returns (T)`
	`/`	`function (T, T) pure returns (T)`
	`%`	`function (T, T) pure returns (T)`
Comparison	`==`	`function (T, T) pure returns (bool)`
	`!=`	`function (T, T) pure returns (bool)`
	`<`	`function (T, T) pure returns (bool)`
	`<=`	`function (T, T) pure returns (bool)`
	`>`	`function (T, T) pure returns (bool)`
	`>=`	`function (T, T) pure returns (bool)`

Note that unary and binary `-` need separate definitions. The compiler will choose the right definition based on how the operator is invoked.

The `using A for B;` directive is active only within the current scope (either the contract or the current module/source unit), including within all of its functions, and has no effect outside of the contract or module in which it is used.

When the directive is used at file level and applied to a user-defined type which was defined at file level in the same file, the word `global` can be added at the end. This will have the effect that the functions and operators are attached to the type everywhere the type is available (including other files), not only in the scope of the using statement.

Let us rewrite the set example from the *Libraries* section in this way, using file-level functions instead of library functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

struct Data { mapping(uint => bool) flags; }
// Now we attach functions to the type.
// The attached functions can be used throughout the rest of the module.
// If you import the module, you have to
// repeat the using directive there, for example as
//   import "flags.sol" as Flags;
//   using {Flags.insert, Flags.remove, Flags.contains}
//     for Flags.Data;
using {insert, remove, contains} for Data;

function insert(Data storage self, uint value)
    returns (bool)
{
    if (self.flags[value])
        return false; // already there
    self.flags[value] = true;
    return true;
}

function remove(Data storage self, uint value)
    returns (bool)
{
    if (!self.flags[value])
        return false; // not there
    self.flags[value] = false;
    return true;
}

function contains(Data storage self, uint value)
    view
    returns (bool)
{
    return self.flags[value];
}


contract C {
    Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
```

(continues on next page)

(continued from previous page)

```
        require(knownValues.insert(value));
    }
}
```

It is also possible to extend built-in types in that way. In this example, we will use a library.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}
using Search for uint[];

contract C {
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint from, uint to) public {
        // This performs the library function call
        uint index = data.indexOf(from);
        if (index == type(uint).max)
            data.push(to);
        else
            data[index] = to;
    }
}
```

Note that all external library calls are actual EVM function calls. This means that if you pass memory or value types, a copy will be performed, even in case of the `self` variable. The only situation where no copy will be performed is when storage reference variables are used or when internal library functions are called.

Another example shows how to define a custom operator for a user-defined type:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.19;

type UFixed16x2 is uint16;

using {
    add as +,
    div as /
```

(continues on next page)

(continued from previous page)

```
} for UFixed16x2 global;

uint32 constant SCALE = 100;

function add(UFixed16x2 a, UFixed16x2 b) pure returns (UFixed16x2) {
    return UFixed16x2.wrap(UFixed16x2.unwrap(a) + UFixed16x2.unwrap(b));
}

function div(UFixed16x2 a, UFixed16x2 b) pure returns (UFixed16x2) {
    uint32 a32 = UFixed16x2.unwrap(a);
    uint32 b32 = UFixed16x2.unwrap(b);
    uint32 result32 = a32 * SCALE / b32;
    require(result32 <= type(uint16).max, "Divide overflow");
    return UFixed16x2.wrap(uint16(a32 * SCALE / b32));
}

contract Math {
    function avg(UFixed16x2 a, UFixed16x2 b) public pure returns (UFixed16x2) {
        return (a + b) / UFixed16x2.wrap(200);
    }
}
```

3.10 Inline Assembly

You can interleave Solidity statements with inline assembly in a language close to the one of the Ethereum Virtual Machine. This gives you more fine-grained control, which is especially useful when you are enhancing the language by writing libraries or optimizing gas usage.

The language used for inline assembly in Solidity is called *Yul* and it is documented in its own section. This section will only cover how the inline assembly code can interface with the surrounding Solidity code.

Warning

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

An inline assembly block is marked by `assembly { ... }`, where the code inside the curly braces is code in the *Yul* language.

The inline assembly code can access local Solidity variables as explained below.

Different inline assembly blocks share no namespace, i.e. it is not possible to call a Yul function or access a Yul variable defined in a different inline assembly block.

3.10.1 Example

The following example provides library code to access the code of another contract and load it into a `bytes` variable. This is possible with "plain Solidity" too, by using `<address>.code`. But the point here is that reusable assembly libraries can enhance the Solidity language without a compiler change.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library GetCode {
    // This will report a warning - `at` will be promoted to reserved keyword
    function at(address addr) public view returns (bytes memory code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(addr)
            // allocate output byte array - this could also be done without assembly
            // by using code = new bytes(size)
            code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(addr, add(code, 0x20), 0, size)
        }
    }
}
```

Inline assembly is also beneficial in cases where the optimizer fails to produce efficient code, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;


library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i)
            sum += data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Same as above, but accomplish the entire code within inline assembly.
    function sumPureAsm(uint[] memory data) public pure returns (uint sum) {
        assembly {
            // Load the length (first 32 bytes)
            let len := mload(data)
```

(continues on next page)

(continued from previous page)

```
            // Skip over the length field.
            //
            // Keep temporary variable so it can be incremented in place.
            //
            // NOTE: incrementing data would result in an unusable
            //       data variable after this assembly block
            let dataElementLocation := add(data, 0x20)

            // Iterate until the bound is not met.
            for
                { let end := add(dataElementLocation, mul(len, 0x20)) }
                lt(dataElementLocation, end)
                { dataElementLocation := add(dataElementLocation, 0x20) }
            {
                sum := add(sum, mload(dataElementLocation))
            }
        }
    }
}
```

3.10.2 Access to External Variables, Functions and Libraries

You can access Solidity variables and other identifiers by using their name.

Local variables of value type are directly usable in inline assembly. They can both be read and assigned to.

Local variables that refer to memory evaluate to the address of the variable in memory, not the value itself. Such variables can also be assigned to, but note that an assignment will only change the pointer and not the data and that it is your responsibility to respect Solidity's memory management. See *Conventions in Solidity*.

Similarly, local variables that refer to statically-sized calldata arrays or calldata structs evaluate to the address of the variable in calldata, not the value itself. The variable can also be assigned a new offset, but note that no validation is performed to ensure that the variable will not point beyond `calldatasize()`.

For external function pointers the address and the function selector can be accessed using `x.address` and `x.selector`. The selector consists of four right-aligned bytes. Both values can be assigned to. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

contract C {
    // Assigns a new selector and address to the return variable @fun
    function combineToFunctionPointer(address newAddress, uint newSelector) public pure␣
↪returns (function() external fun) {
        assembly {
            fun.selector := newSelector
            fun.address  := newAddress
        }
    }
}
```

For dynamic calldata arrays, you can access their calldata offset (in bytes) and length (number of elements) using `x.offset` and `x.length`. Both expressions can also be assigned to, but as for the static case, no validation will be performed to ensure that the resulting data area is within the bounds of `calldatasize()`.

For local storage variables or state variables (including transient storage) a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, their "address" is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x.slot`, and to retrieve the byte-offset you use `x.offset`. Using `x` itself will result in an error.

You can also assign to the `.slot` part of a local storage variable pointer. For these (structs, arrays or mappings), the `.offset` part is always zero. It is not possible to assign to the `.slot` or `.offset` part of a state variable, though.

Local Solidity variables are available for assignments, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.28 <0.9.0;

// This will report a warning
contract C {
    bool transient a;
    uint b;
    function f(uint x) public returns (uint r) {
        assembly {
            // We ignore the storage slot offset, we know it is zero
            // in this special case.
            r := mul(x, sload(b.slot))
            tstore(a.slot, true)
        }
    }
}
```

Warning

If you access variables of a type that spans less than 256 bits (for example `uint64`, `address`, or `bytes16`), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important: `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` To clean signed types, you can use the `signextend` opcode: `assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

Since Solidity 0.6.0, the name of an inline assembly variable may not shadow any declaration visible in the scope of the inline assembly block (including variable, contract and function declarations).

Since Solidity 0.7.0, variables and functions declared inside the inline assembly block may not contain `.`, but using `.` is valid to access Solidity variables from outside the inline assembly block. However, it is still valid to use dots if you use Solidity in Yul-only mode.

3.10.3 Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

3.10.4 Conventions in Solidity

Values of Typed Variables

In contrast to EVM assembly, Solidity has types which are narrower than 256 bits, e.g. `uint24`. For efficiency, most arithmetic operations ignore the fact that types can be shorter than 256 bits, and the higher-order bits are cleaned when

necessary, i.e., shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher-order bits first.

Memory Management

Solidity manages memory in the following way. There is a "free memory pointer" at position `0x40` in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Solidity does not guarantee and does not require that the values in memory are placed at positions aligned to a multiple of any value. Here is an assembly snippet you can use for allocating memory that follows the process outlined above:

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

The first 64 bytes of memory can be used as "scratch space" for short-term allocation. The 32 bytes after the free memory pointer (i.e., starting at `0x60`) are meant to be zero permanently and is used as the initial value for empty dynamic memory arrays. This means that the allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `bytes1[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

Warning

Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically and dynamically-sized arrays; so, do not rely on this.

Memory Safety

Without the use of inline assembly, the compiler can rely on memory to remain in a well-defined state at all times. This is especially relevant for *the new code generation pipeline via Yul IR*: this code generation path can move local variables from stack to memory to avoid stack-too-deep errors and perform additional memory optimizations, if it can rely on certain assumptions about memory use.

While we recommend to always respect Solidity's memory model, inline assembly allows you to use memory in an incompatible way. Therefore, moving stack variables to memory and additional memory optimizations are, by default, globally disabled in the presence of any inline assembly block that contains a memory operation or assigns to Solidity variables in memory.

However, you can specifically annotate an assembly block to indicate that it in fact respects Solidity's memory model as follows:

```
assembly ("memory-safe") {
    ...
}
```

In particular, a memory-safe assembly block may only access the following memory ranges:

- Memory allocated by yourself using a mechanism like the `allocate` function described above.
- Memory allocated by Solidity, e.g. memory within the bounds of a memory array you reference.
- The scratch space between memory offset 0 and 64 mentioned above.

- Temporary memory that is located *after* the value of the free memory pointer at the beginning of the assembly block, i.e. memory that is "allocated" at the free memory pointer without updating the free memory pointer.

Furthermore, if the assembly block assigns to Solidity variables in memory, you need to assure that accesses to the Solidity variables only access these memory ranges.

Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates. As an example, the following assembly snippet is not memory safe, because the value of `returndatasize()` may exceed the 64 byte scratch space:

```
assembly {
  returndatacopy(0, 0, returndatasize())
  revert(0, returndatasize())
}
```

On the other hand, the following code *is* memory safe, because memory beyond the location pointed to by the free memory pointer can safely be used as temporary scratch space:

```
assembly ("memory-safe") {
  let p := mload(0x40)
  returndatacopy(p, 0, returndatasize())
  revert(p, returndatasize())
}
```

Note that you do not need to update the free memory pointer if there is no following allocation, but you can only use memory starting from the current offset given by the free memory pointer.

If the memory operations use a length of zero, it is also fine to just use any offset (not only if it falls into the scratch space):

```
assembly ("memory-safe") {
  revert(0, 0)
}
```

Note that not only memory operations in inline assembly itself can be memory-unsafe, but also assignments to Solidity variables of reference type in memory. For example the following is not memory-safe:

```
bytes memory x;
assembly {
  x := 0x40
}
x[0x20] = 0x42;
```

Inline assembly that neither involves any operations that access memory nor assigns to any Solidity variables in memory is automatically considered memory-safe and does not need to be annotated.

Warning

It is your responsibility to make sure that the assembly actually satisfies the memory model. If you annotate an assembly block as memory-safe, but violate one of the memory assumptions, this **will** lead to incorrect and undefined behavior that cannot easily be discovered by testing.

In case you are developing a library that is meant to be compatible across multiple versions of Solidity, you can use a special comment to annotate an assembly block as memory-safe:

```
/// @solidity memory-safe-assembly
assembly {
    ...
}
```

Warning

The `memory-safe-assembly` special comment is deprecated and scheduled for removal. In new code targeting recent compilers, use the assembly block annotation.

3.10.5 Advanced Safe Use of Memory

Beyond the strict definition of memory-safety given above, there are cases in which you may want to use more than 64 bytes of scratch space starting at memory offset `0`. If you are careful, it can be admissible to use memory up to (and not including) offset `0x80` and still safely declare the assembly block as `memory-safe`. This is admissible under either of the following conditions:

- By the end of the assembly block, the free memory pointer at offset `0x40` is restored to a sane value (i.e. it is either restored to its original value or an increment of it due to a manual memory allocation), and the memory word at offset `0x60` is restored to a value of zero.
- The assembly block terminates, i.e. execution can never return to high-level Solidity code. This is the case, for example, if your assembly block unconditionally ends in calling the `revert` opcode.

Furthermore, you need to be aware that the default-value of dynamic arrays in Solidity point to memory offset `0x60`, so for the duration of temporarily changing the value at memory offset `0x60`, you can no longer rely on getting accurate length values when reading dynamic arrays, until you restore the zero value at `0x60`. To be more precise, we only guarantee safety when overwriting the zero pointer, if the remainder of the assembly snippet does not interact with the memory of high-level Solidity objects (including by reading from offsets previously stored in variables).

3.11 Cheatsheet

3.11.1 Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	++, --
	New expression	new <typename>
	Array subscripting	<array>[<index>]
	Member access	<object>.<member>
	Function-like call	<func>(<args...>)
	Parentheses	(<statement>)
2	Prefix increment and decrement	++, --
	Unary minus	-
	Unary operations	delete
	Logical NOT	!
	Bitwise NOT	~
3	Exponentiation	**
4	Multiplication, division and modulo	*, /, %
5	Addition and subtraction	+, -
6	Bitwise shift operators	<<, >>
7	Bitwise AND	&
8	Bitwise XOR	^
9	Bitwise OR	|
10	Inequality operators	<, >, <=, >=
11	Equality operators	==, !=
12	Logical AND	&&
13	Logical OR	||
14	Ternary operator	<conditional> ? <if-true> : <if-false>
	Assignment operators	=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
15	Comma operator	,

3.11.2 ABI Encoding and Decoding Functions

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: *ABI*-decodes the provided data. The types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: *ABI*-encodes the given arguments
- `abi.encodePacked(...) returns (bytes memory)`: Performs *packed encoding* of the given arguments. Note that this encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: *ABI*-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`: ABI-encodes a call to `functionPointer` with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals `abi.encodeWithSelector(functionPointer.selector, ...)`
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`

3.11.3 Members of `bytes` and `string`

- `bytes.concat(...) returns (bytes memory)`: *Concatenates variable number of arguments to one byte array*

- `string.concat(...) returns (string memory)`: *Concatenates variable number of arguments to one string array*

3.11.4 Members of `address`

- `<address>.balance` (`uint256`): balance of the *Address* in Wei
- `<address>.code` (`bytes memory`): code at the *Address* (can be empty)
- `<address>.codehash` (`bytes32`): the codehash of the *Address*
- `<address>.call(bytes memory) returns (bool, bytes memory)`: issue low-level `CALL` with the given payload, returns success condition and return data
- `<address>.delegatecall(bytes memory) returns (bool, bytes memory)`: issue low-level `DELEGATECALL` with the given payload, returns success condition and return data
- `<address>.staticcall(bytes memory) returns (bool, bytes memory)`: issue low-level `STATICCALL` with the given payload, returns success condition and return data
- `<address payable>.send(uint256 amount) returns (bool)`: send given amount of Wei to *Address*, returns `false` on failure (deprecated)
- `<address payable>.transfer(uint256 amount)`: send given amount of Wei to *Address*, throws on failure (deprecated)

3.11.5 Block and Transaction Properties

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `blobhash(uint index) returns (bytes32)`: versioned hash of the `index`-th blob associated with the current transaction. A versioned hash consists of a single byte representing the version (currently `0x01`), followed by the last 31 bytes of the SHA256 hash of the KZG commitment (EIP-4844). Returns zero if no blob with the given index exists.
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.blobbasefee` (`uint`): current block's blob base fee (EIP-7516 and EIP-4844)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty (`EVM < Paris`). For other EVM versions it behaves as a deprecated alias for `block.prevrandao` that will be removed in the next breaking release
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.prevrandao` (`uint`): random number provided by the beacon chain (`EVM >= Paris`) (see EIP-4399)
- `block.timestamp` (`uint`): current block timestamp in seconds since Unix epoch
- `gasleft() returns (uint256)`: remaining gas
- `msg.data` (`bytes`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `tx.gasprice` (`uint`): gas price of the transaction

- `tx.origin` (`address`): sender of the transaction (full call chain)

3.11.6 Validations and Assertions

- `assert(bool condition)`: abort execution and revert state changes if condition is `false` (use for internal error)
- `require(bool condition)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
- `require(bool condition, string memory message)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
- `revert()`: abort execution and revert state changes
- `revert(string memory message)`: abort execution and revert state changes providing an explanatory string

3.11.7 Mathematical and Cryptographic Functions

- `keccak256(bytes memory) returns (bytes32)`: compute the Keccak-256 hash of the input
- `sha256(bytes memory) returns (bytes32)`: compute the SHA-256 hash of the input
- `ripemd160(bytes memory) returns (bytes20)`: compute the RIPEMD-160 hash of the input
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover address associated with the public key from elliptic curve signature, return zero on error
- `addmod(uint x, uint y, uint k) returns (uint)`: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `mulmod(uint x, uint y, uint k) returns (uint)`: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `erc7201(string memory id) returns (uint)`: compute the base slot of an `erc7201` storage namespace. Can be used in compile time context.

3.11.8 Contract-related

- `this` (current contract's type): the current contract, explicitly convertible to `address` or `address payable`
- `super`: a contract one level higher in the inheritance hierarchy
- `selfdestruct(address payable recipient)`: send all funds to the given address and (only on EVMs before Cancun or when invoked within the transaction creating the contract) destroy the contract.

3.11.9 Type Information

- `type(C).name` (`string`): the name of the contract
- `type(C).creationCode` (`bytes memory`): creation bytecode of the given contract, see *Type Information*.
- `type(C).runtimeCode` (`bytes memory`): runtime bytecode of the given contract, see *Type Information*.
- `type(I).interfaceId` (`bytes4`): value containing the EIP-165 interface identifier of the given interface, see *Type Information*.
- `type(T).min` (`T`): the minimum value representable by the integer type `T`, see *Type Information*.
- `type(T).max` (`T`): the maximum value representable by the integer type `T`, see *Type Information*.

3.11.10 Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externally and internally (creates a *getter function* for storage/state variables)
- `private`: only visible in the current contract
- `external`: only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- `internal`: only visible internally

3.11.11 Modifiers

- `pure` for functions: Disallows modification or access of state.
- `view` for functions: Disallows modification of state.
- `payable` for functions: Allows them to receive Ether together with a call.
- `constant` for state variables: Disallows assignment (except initialization), does not occupy storage slot.
- `immutable` for state variables: Allows assignment at construction time and is constant when deployed. Is stored in code.
- `anonymous` for events: Does not store event signature as topic.
- `indexed` for event parameters: Stores the parameter as topic.
- `virtual` for functions and modifiers: Allows the function's or modifier's behavior to be changed in derived contracts.
- `override`: States that this function, modifier or public state variable changes the behavior of a function or modifier in a base contract.

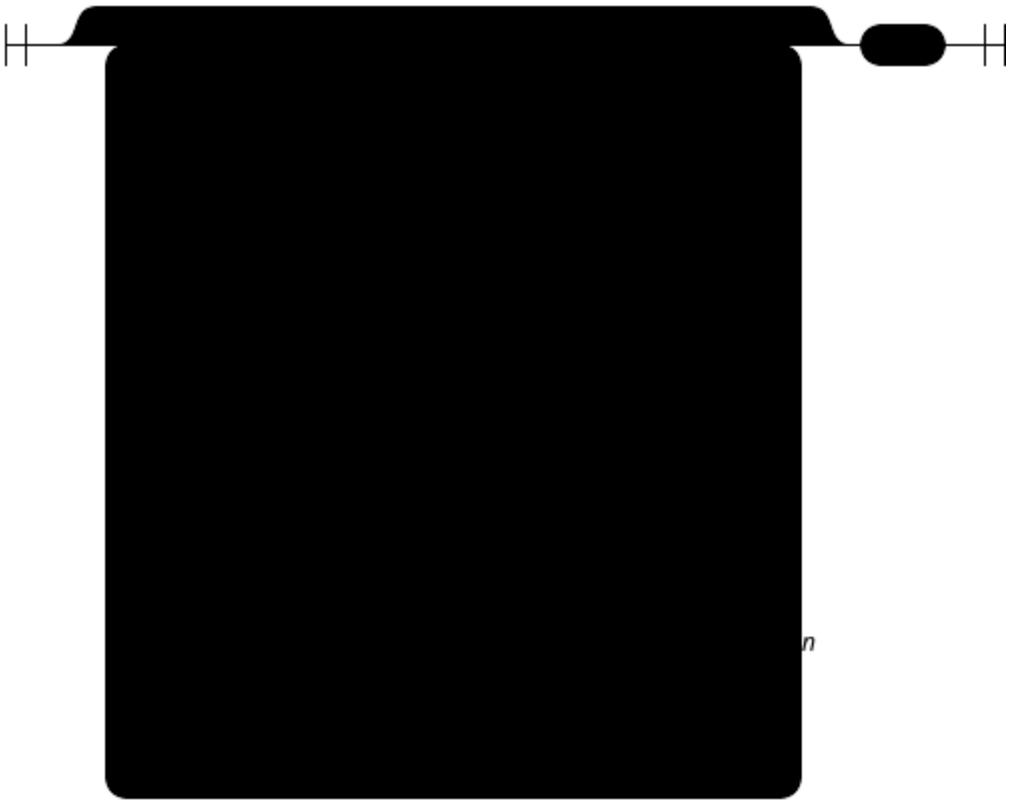
3.12 Language Grammar

grammar SolidityParser

Solidity is a statically typed, contract-oriented, high-level language for implementing smart contracts on the Ethereum platform.

source-unit

On top level, Solidity allows pragmas, import directives, and definitions of contracts, interfaces, libraries, structs, enums and constants.



`import-directive`

Import directives import identifiers from different files.

`path`

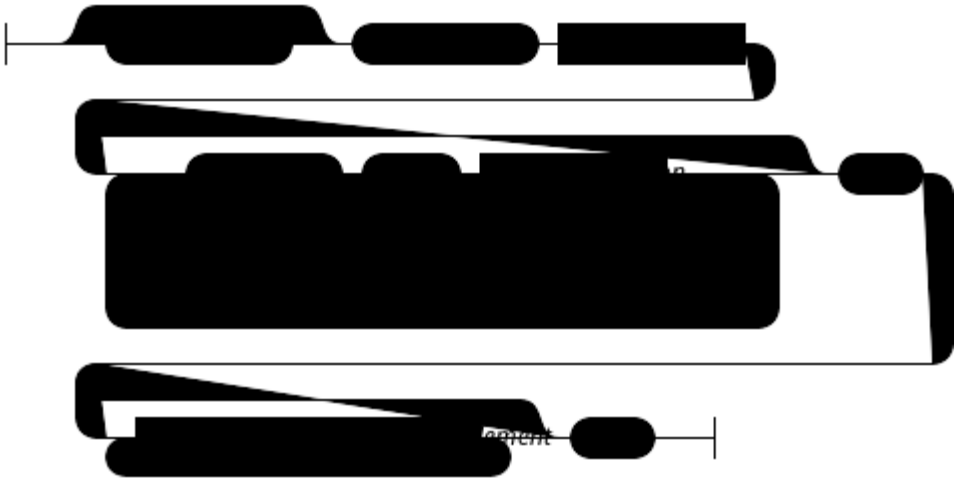
Path of a file to be imported.

`symbol-aliases`

List of aliases for symbols to be imported.

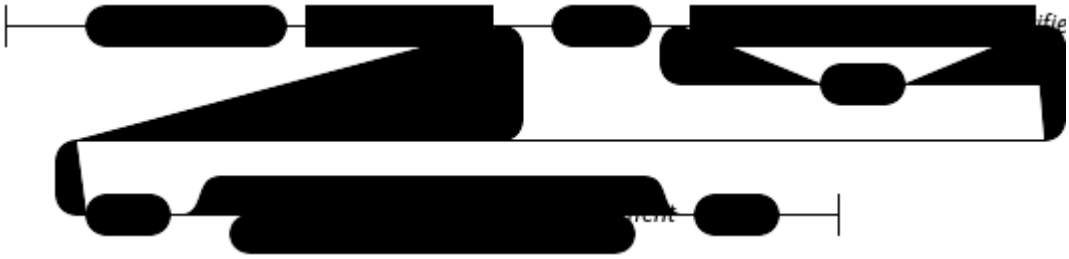
`contract-definition`

Top-level definition of a contract.



interface-definition

Top-level definition of an interface.



library-definition

Top-level definition of a library.

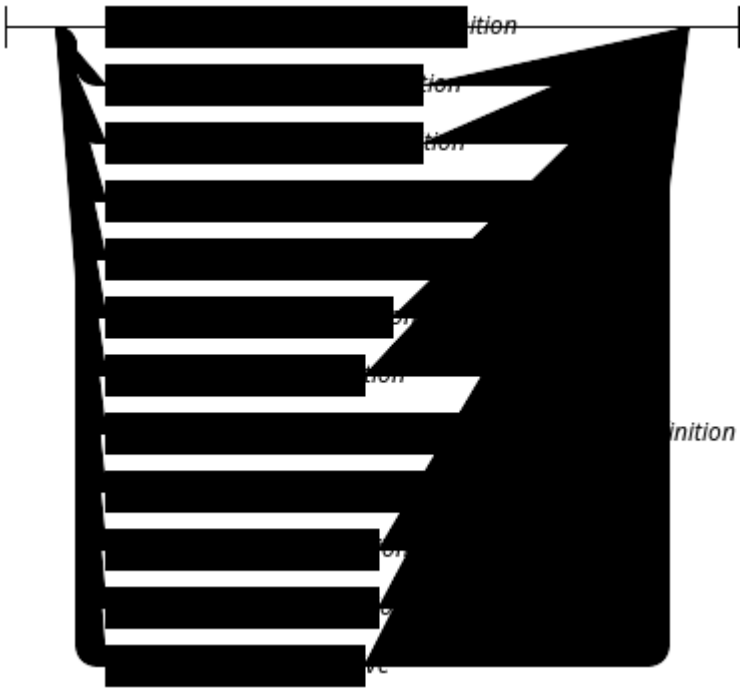
inheritance-specifier

Inheritance specifier for contracts and interfaces. Can optionally supply base constructor arguments.

contract-body-element

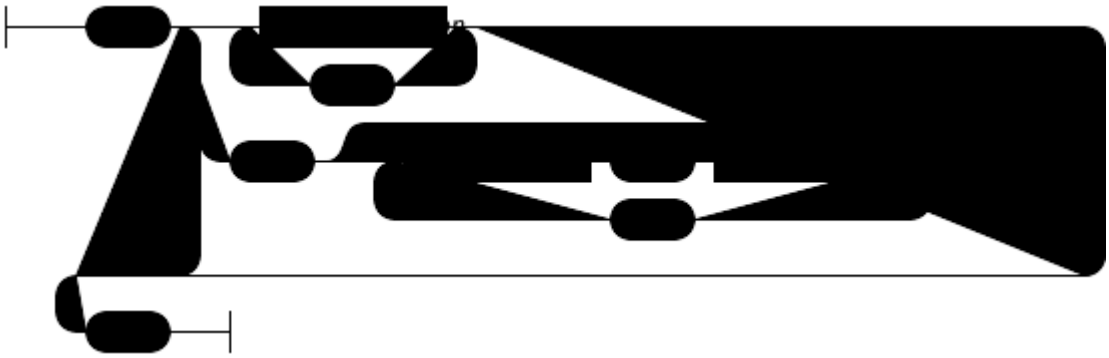
Declarations that can be used in contracts, interfaces and libraries.

Note that interfaces and libraries may not contain constructors, interfaces may not contain state variables and libraries may not contain fallback, receive functions nor non-constant state variables.



`call-argument-list`

Arguments when calling a function or a similar callable object. The arguments are either given as comma separated list or as map of named arguments.



`identifier-path`

Qualified name.

`modifier-invocation`

Call to a modifier. If the modifier takes no arguments, the argument list can be skipped entirely (including opening and closing parentheses).

`visibility`

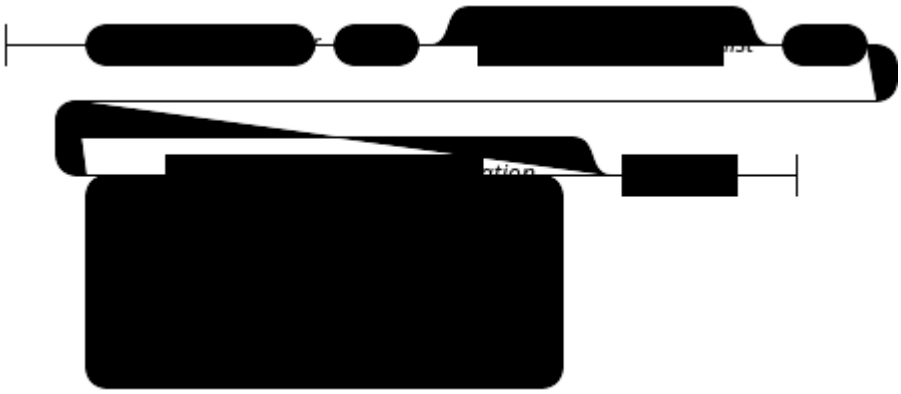
Visibility for functions and function types.

parameter-list

A list of parameters, such as function arguments or return values.

constructor-definition

Definition of a constructor. Must always supply an implementation. Note that specifying internal or public visibility is deprecated.



state-mutability

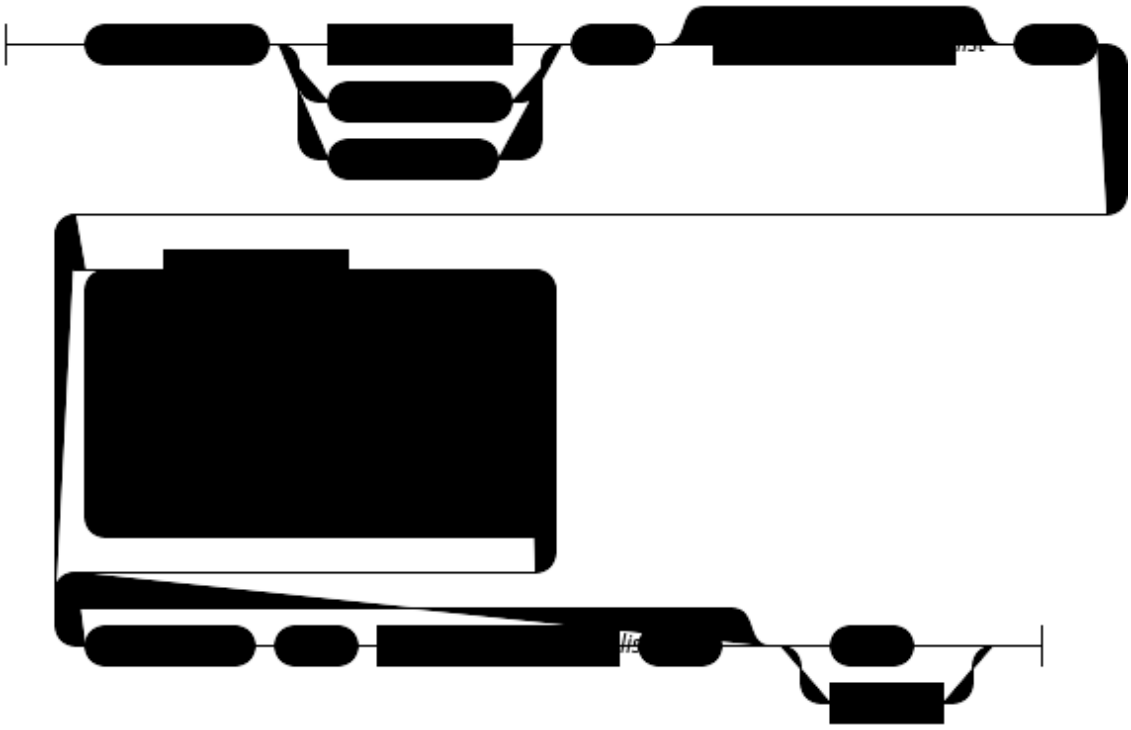
State mutability for function types. The default mutability 'non-payable' is assumed if no mutability is specified.

override-specifier

An override specifier used for functions, modifiers or state variables. In cases where there are ambiguous declarations in several base contracts being overridden, a complete list of base contracts has to be given.

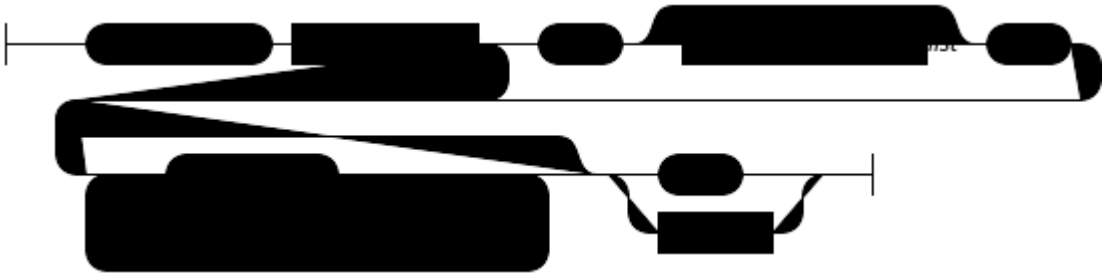
function-definition

The definition of contract, library, interface or free functions. Depending on the context in which the function is defined, further restrictions may apply, e.g. functions in interfaces have to be unimplemented, i.e. may not contain a body block.



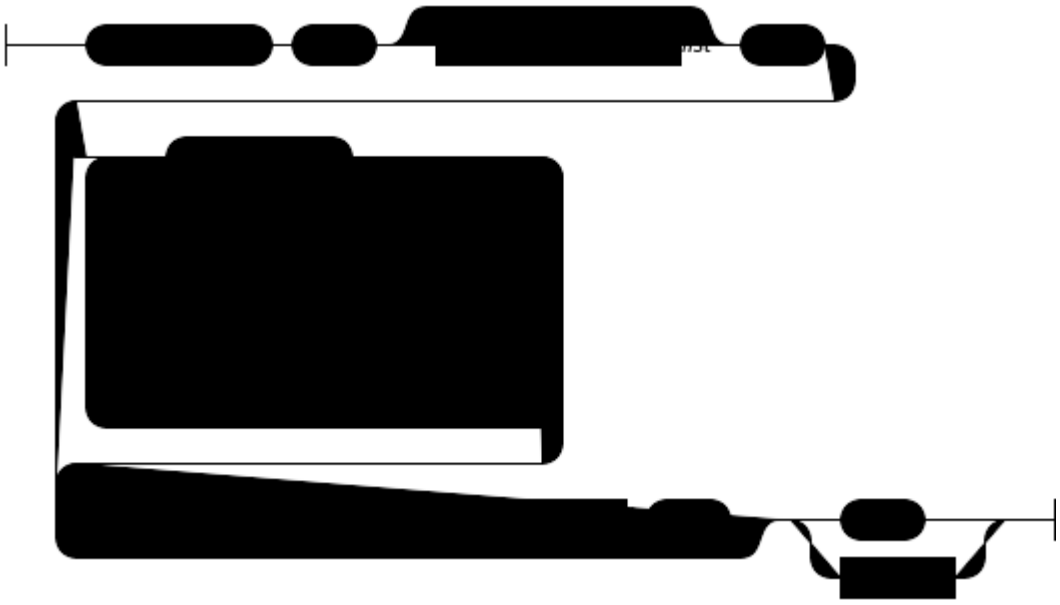
`modifier-definition`

The definition of a modifier. Note that within the body block of a modifier, the underscore cannot be used as identifier, but is used as placeholder statement for the body of a function to which the modifier is applied.



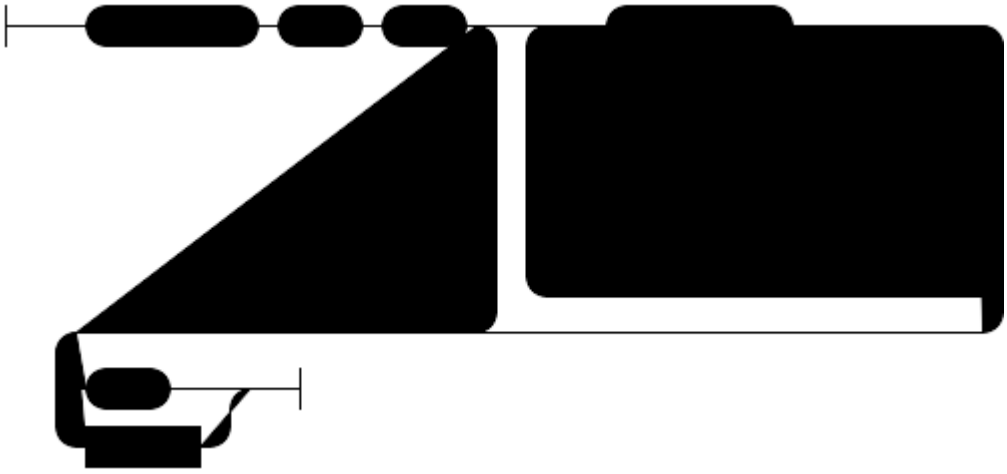
`fallback-function-definition`

Definition of the special fallback function.



receive-function-definition

Definition of the special receive function.



struct-definition

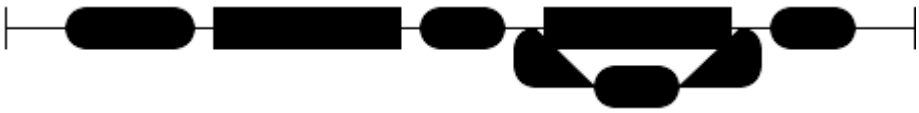
Definition of a struct. Can occur at top-level within a source unit or within a contract, library or interface.

struct-member

The declaration of a named struct member.

enum-definition

Definition of an enum. Can occur at top-level within a source unit or within a contract, library or interface.

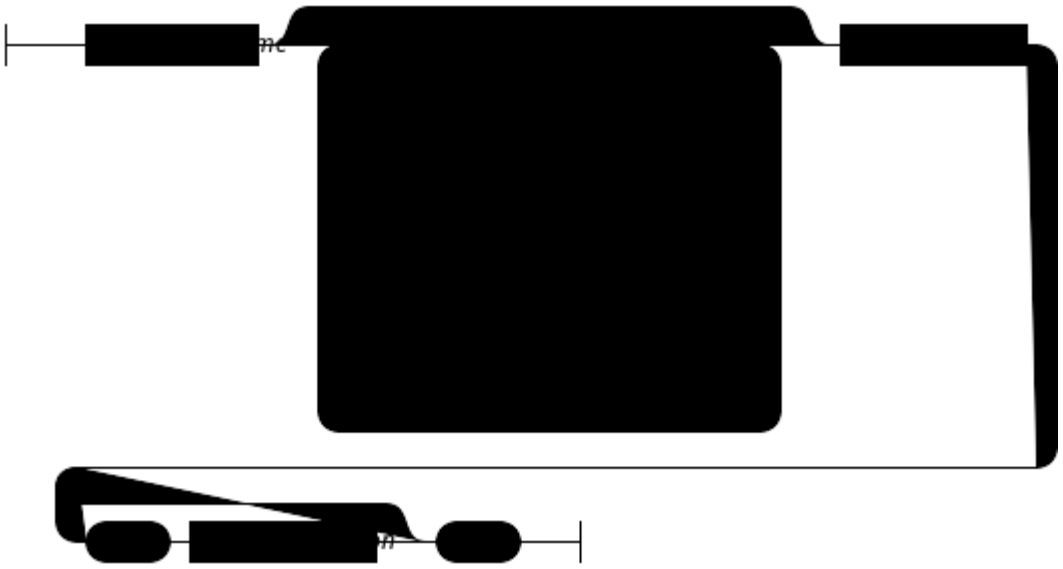


user-defined-value-type-definition

Definition of a user defined value type. Can occur at top-level within a source unit or within a contract, library or interface.

state-variable-declaration

The declaration of a state variable.



constant-variable-declaration

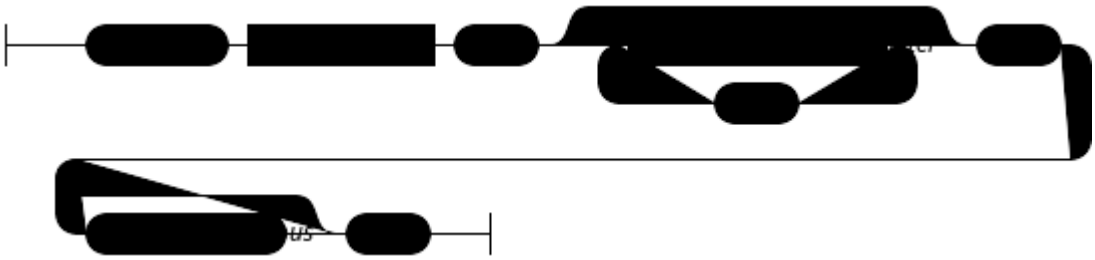
The declaration of a constant variable.

event-parameter

Parameter of an event.

event-definition

Definition of an event. Can occur in contracts, libraries or interfaces.

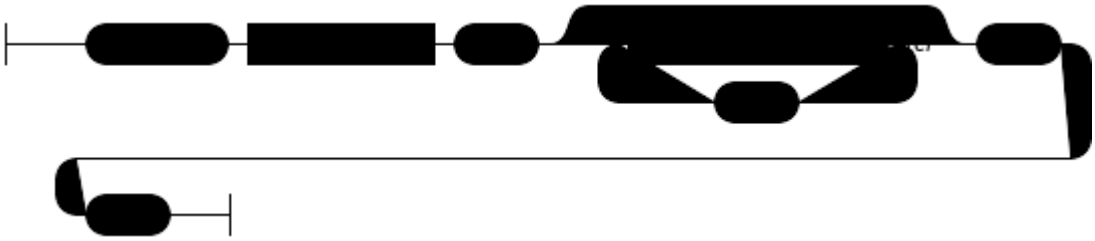


error-parameter

Parameter of an error.

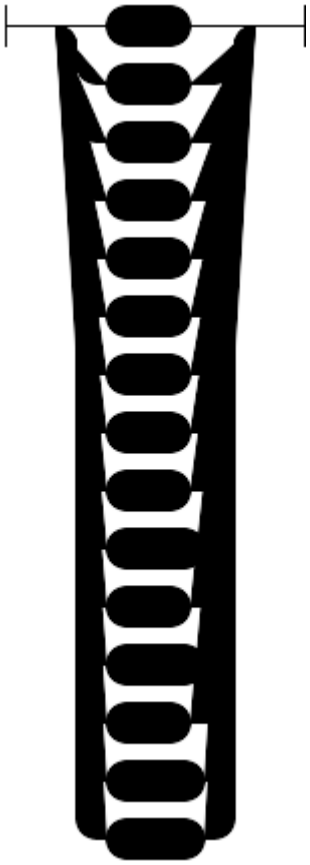
`error-definition`

Definition of an error.



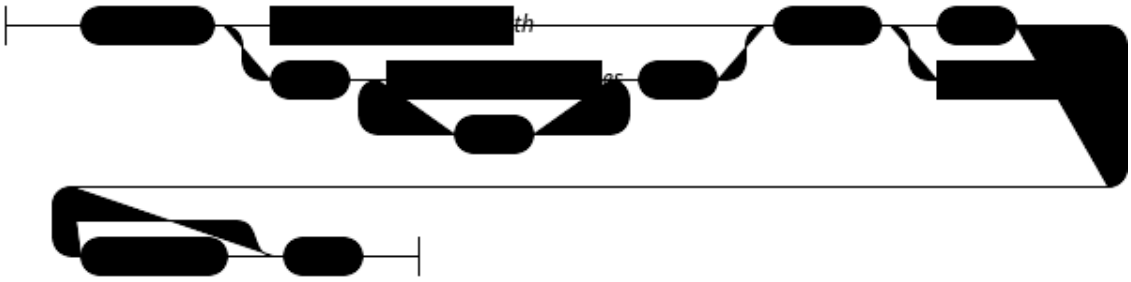
`user-definable-operator`

Operators that users are allowed to implement for some types with *using for*.



`using-directive`

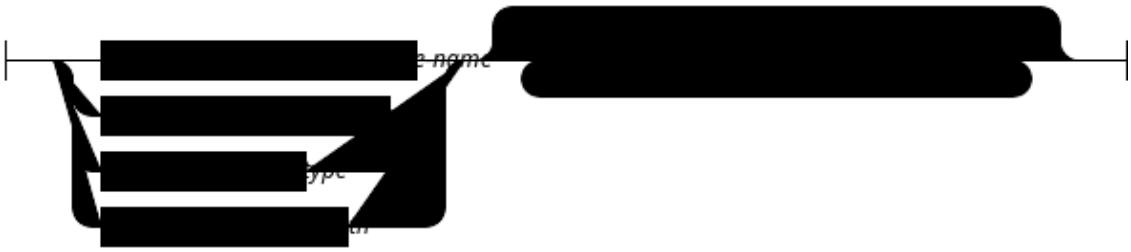
Using directive to attach library functions and free functions to types. Can occur within contracts and libraries and at the file level.



using-aliases

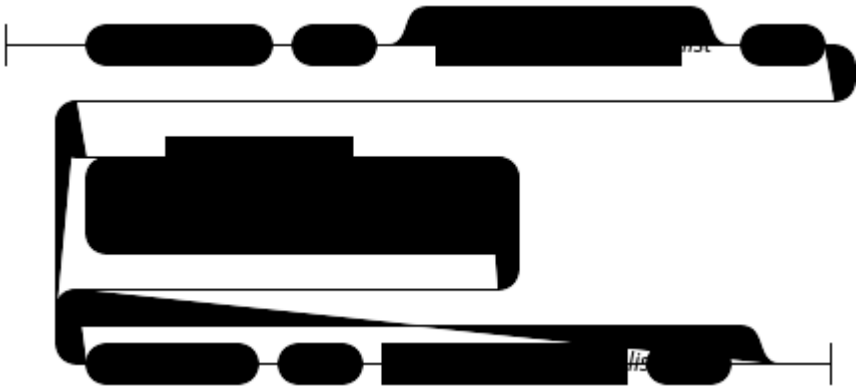
type-name

A type name can be an elementary type, a function type, a mapping type, a user-defined type (e.g. a contract or struct) or an array type.



elementary-type-name

function-type-name



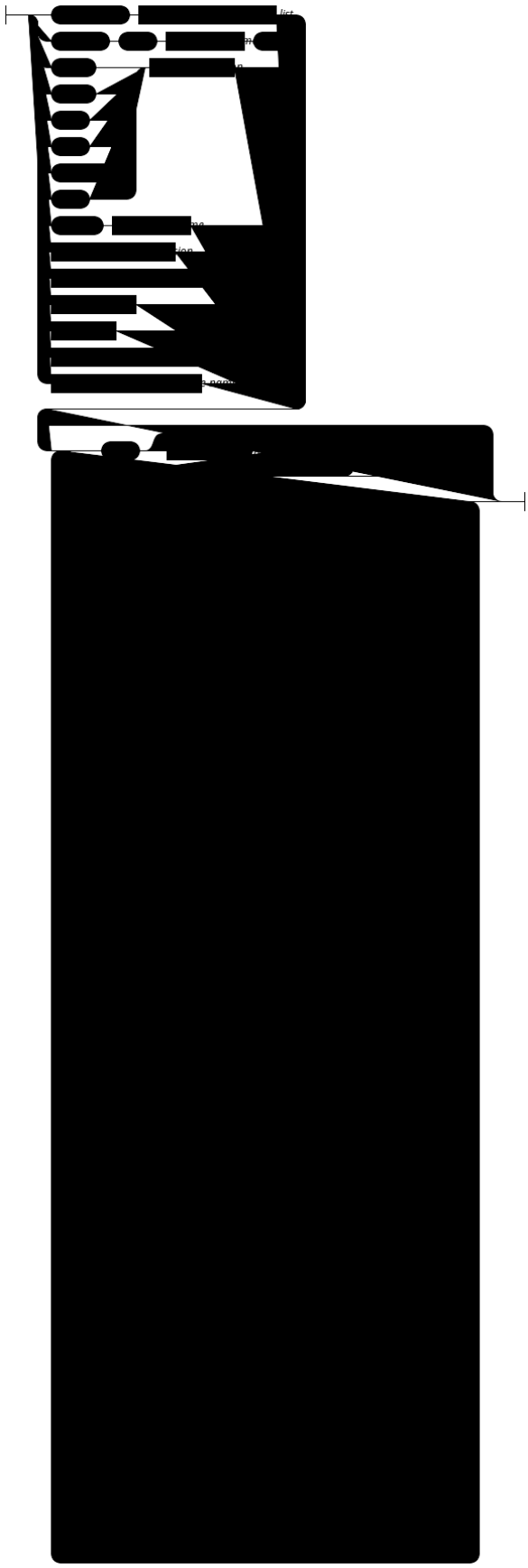
variable-declaration

The declaration of a single variable.

`data-location`

`expression`

Complex expression. Can be an index access, an index range access, a member access, a function call (with optional function call options), a type conversion, an unary or binary expression, a comparison or assignment, a ternary expression, a new-expression (i.e. a contract creation or the allocation of a dynamic memory array), a tuple, an inline array or a primary expression (i.e. an identifier, literal or type name).



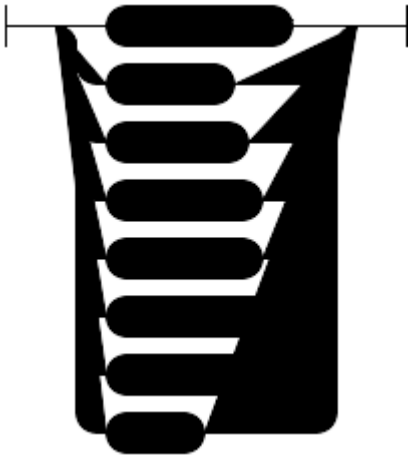
tuple-expression

inline-array-expression

An inline array expression denotes a statically sized array of the common type of the contained expressions.

identifier

Besides regular non-keyword Identifiers, some keywords like ‘from’ and ‘error’ can also be used as identifiers.



literal

literal-with-sub-denomination

boolean-literal

string-literal

A full string literal consists of either one or several consecutive quoted strings.

hex-string-literal

A full hex string literal that consists of either one or several consecutive hex strings.

unicode-string-literal

A full unicode string literal that consists of either one or several consecutive unicode strings.

number-literal

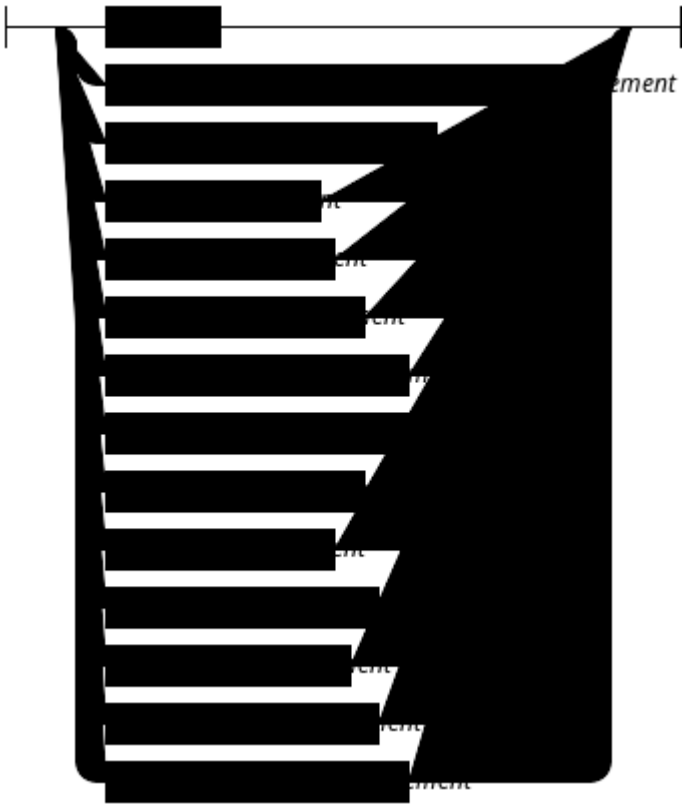
Number literals can be decimal or hexadecimal numbers with an optional unit.

block

A curly-braced block of statements. Opens its own scope.

unchecked-block

statement

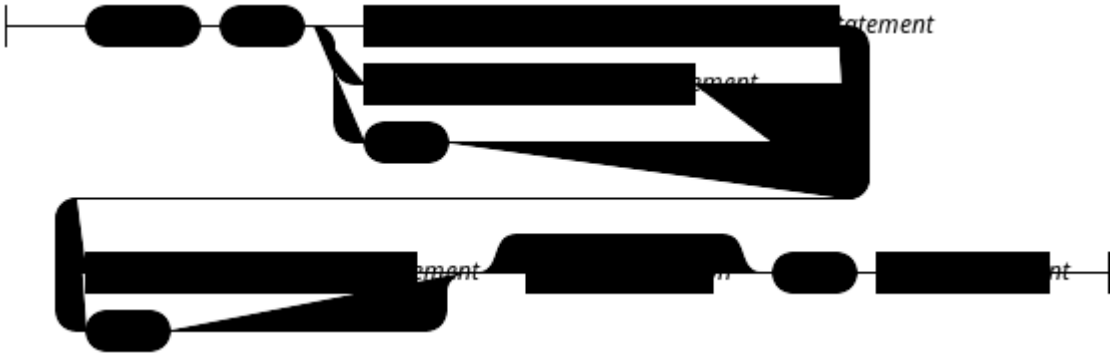


`if-statement`

If statement with optional else part.

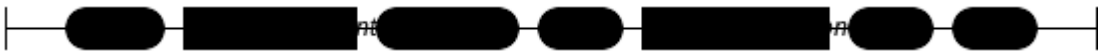
`for-statement`

For statement with optional init, condition and post-loop part.



`while-statement`

`do-while-statement`



continue-statement

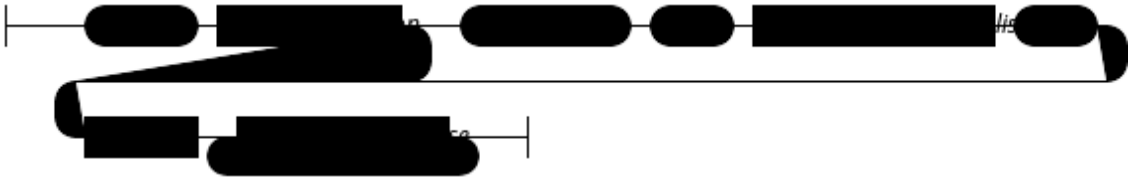
A continue statement. Only allowed inside for, while or do-while loops.

break-statement

A break statement. Only allowed inside for, while or do-while loops.

try-statement

A try statement. The contained expression needs to be an external function call or a contract creation.



catch-clause

The catch clause of a try statement.

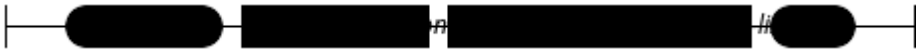
return-statement

emit-statement

An emit statement. The contained expression needs to refer to an event.

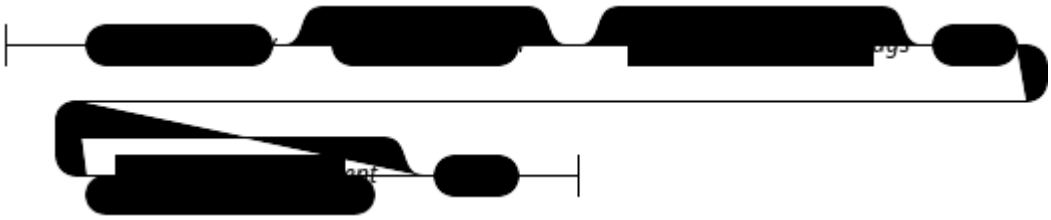
revert-statement

A revert statement. The contained expression needs to refer to an error.



assembly-statement

An inline assembly block. The contents of an inline assembly block use a separate scanner/lexer, i.e. the set of keywords and allowed identifiers is different inside an inline assembly block.



assembly-flags

Assembly flags. Comma-separated list of double-quoted strings as flags.

variable-declaration-tuple

A tuple of variable names to be used in variable declarations. May contain empty fields.

variable-declaration-statement

A variable declaration statement. A single variable may be declared without initial value, whereas a tuple of variables can only be declared with initial value.

expression-statement

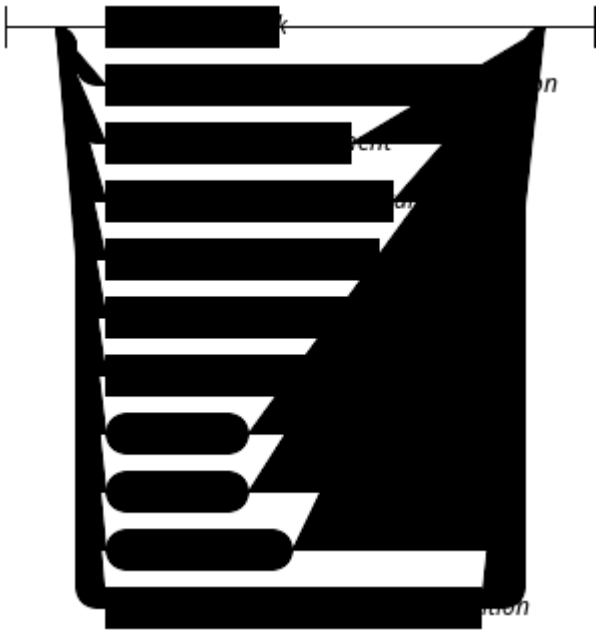
mapping-type

mapping-key-type

Only elementary types or user defined types are viable as mapping keys.

yul-statement

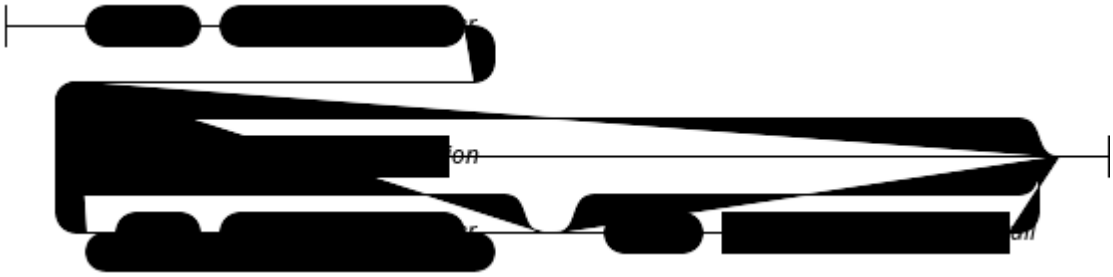
A Yul statement within an inline assembly block. continue and break statements are only valid within for loops. leave statements are only valid within function bodies.



yul-block

yul-variable-declaration

The declaration of one or more Yul variables with optional initial value. If multiple variables are declared, only a function call is a valid initial value.



yul-assignment

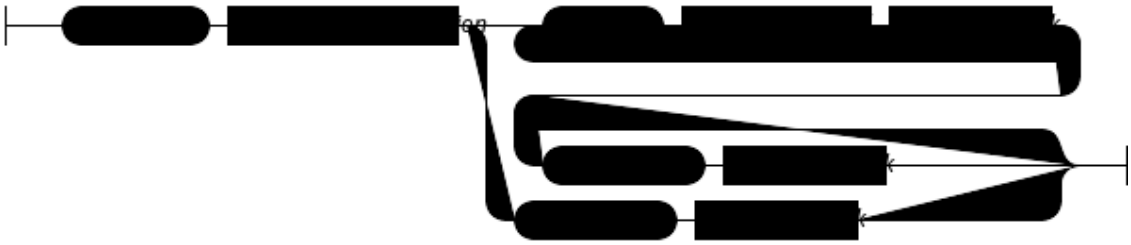
Any expression can be assigned to a single Yul variable, whereas multi-assignments require a function call on the right-hand side.

yul-if-statement

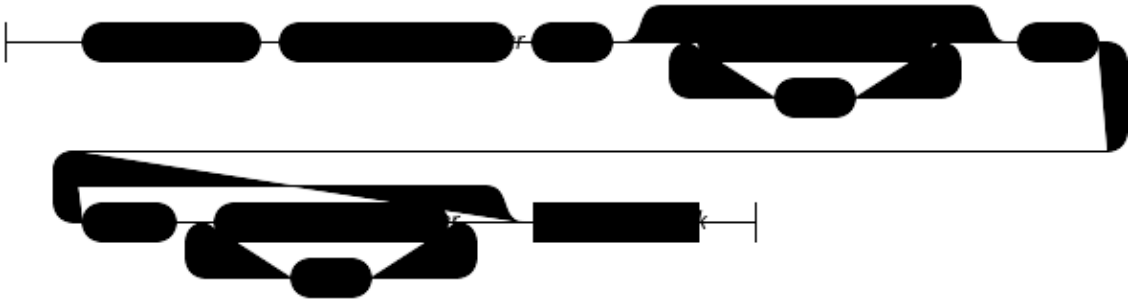
yul-for-statement

yul-switch-statement

A Yul switch statement can consist of only a default-case (deprecated) or one or more non-default cases optionally followed by a default-case.



yul-function-definition



yul-path

While only identifiers without dots can be declared within inline assembly, paths containing dots can refer to declarations outside the inline assembly block.

yul-function-call

A call to a function with return values can only occur as right-hand side of an assignment or a variable declaration.

yul-boolean

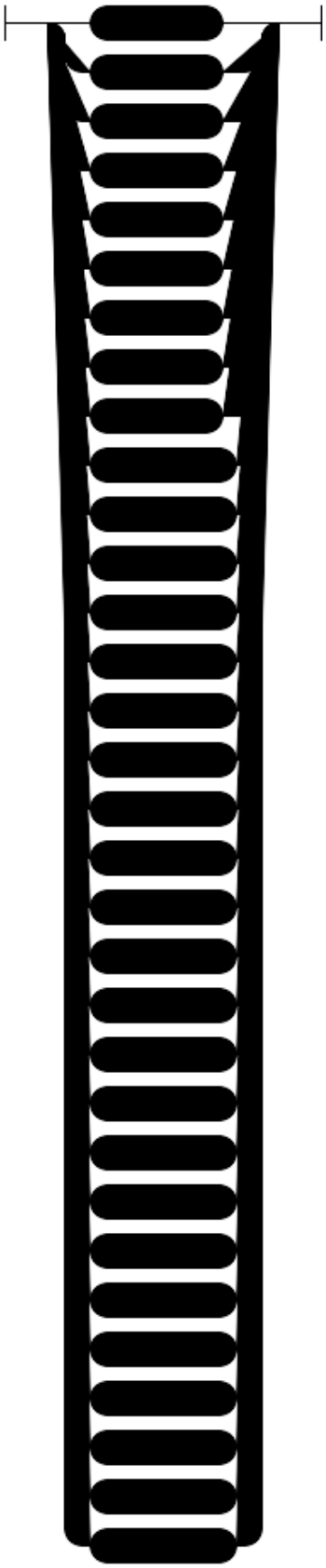
yul-literal

yul-expression

grammar SolidityLexer

fixed-bytes

Bytes types of fixed length.

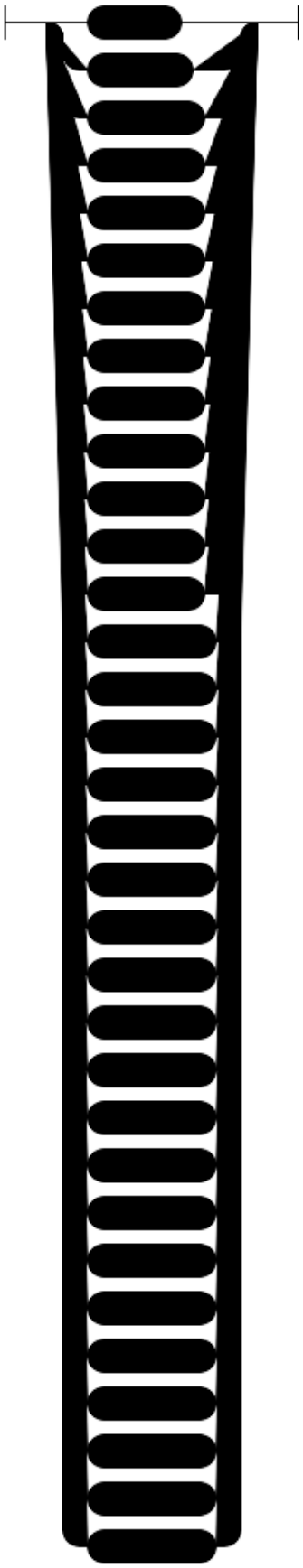


`sub-denomination`

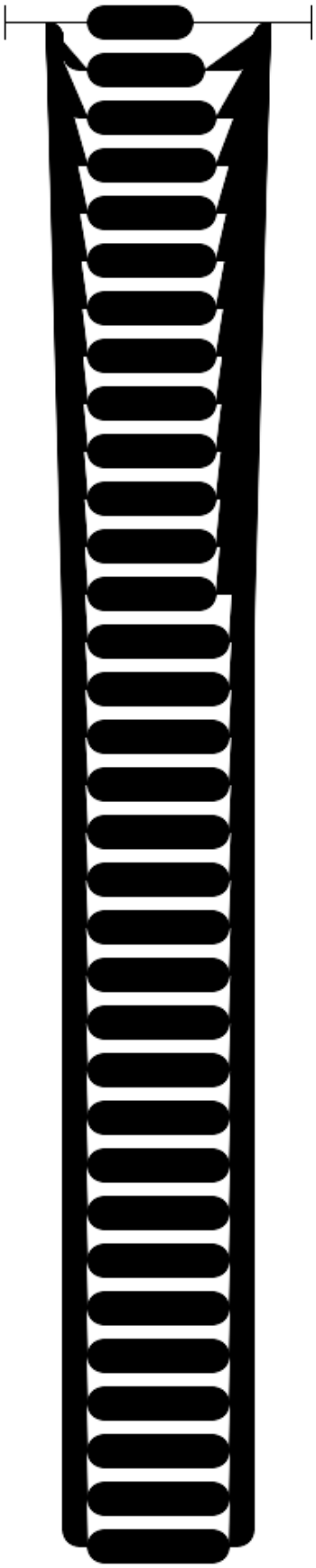
Unit denomination for numbers.

`signed-integer-type`

Sized signed integer types. int is an alias of int256.



`unsigned-integer-type`
: Sized unsigned integer types. uint is an alias of uint256.

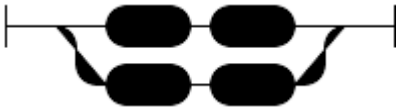


non-empty-string-literal

A non-empty quoted string literal restricted to printable characters.

empty-string-literal

An empty string literal



single-quoted-printable

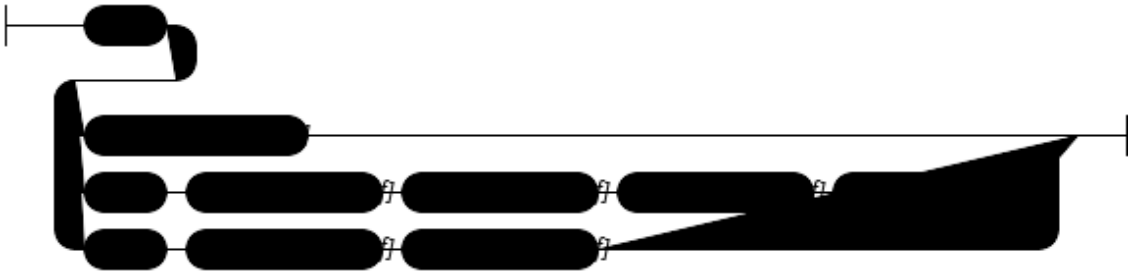
Any printable character except single quote or back slash.

double-quoted-printable

Any printable character except double quote or back slash.

escape-sequence

Escape sequence. Apart from common single character escape sequences, line breaks can be escaped as well as four hex digit unicode escapes \uXXXX and two digit hex escape sequences \xXX are allowed.

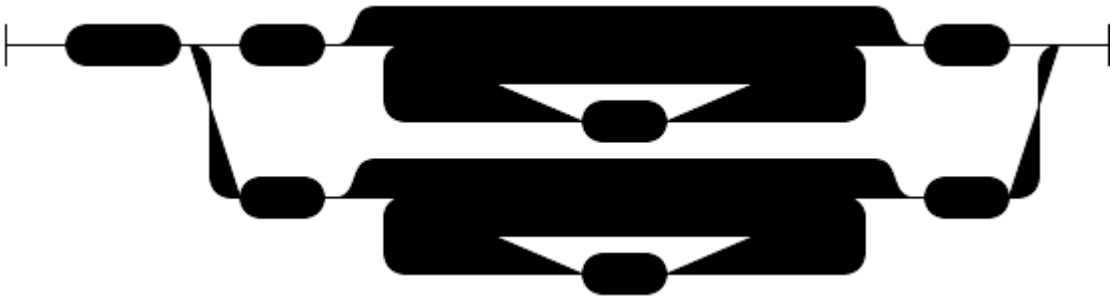


unicode-string-literal

A single quoted string literal allowing arbitrary unicode characters.

hex-string

Hex strings need to consist of an even number of hex digits that may be grouped using underscores.

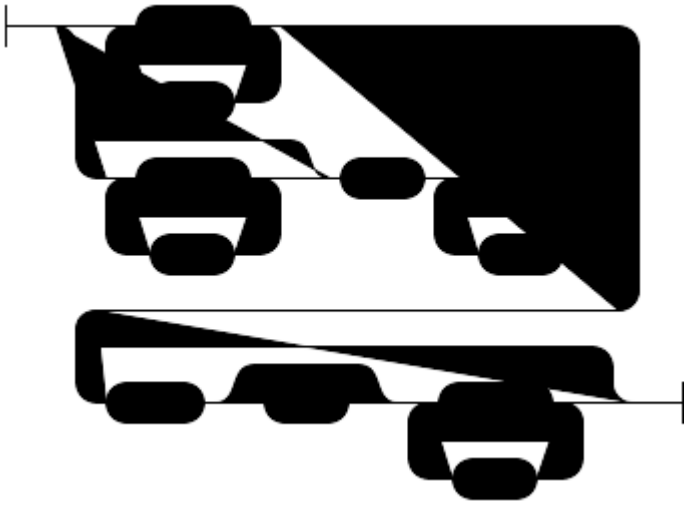


hex-number

Hex numbers consist of a prefix and an arbitrary number of hex digits that may be delimited by underscores.

decimal-number

A decimal number literal consists of decimal digits that may be delimited by underscores and an optional positive or negative exponent. If the digits contain a decimal point, the literal has fixed point type.

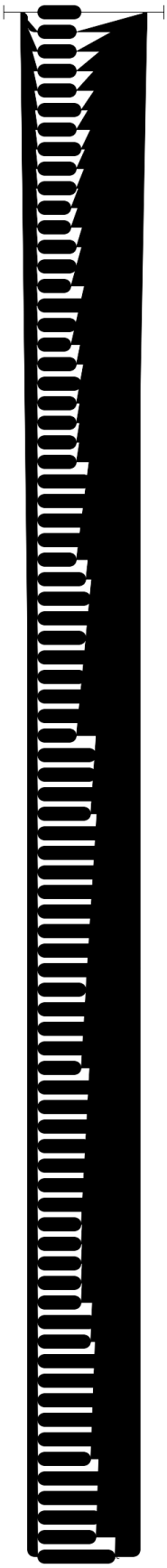


identifier

An identifier in solidity has to start with a letter, a dollar-sign or an underscore and may additionally contain numbers after the first symbol.

yul-evm-builtin

Builtin functions in the EVM Yul dialect.



yul-identifier

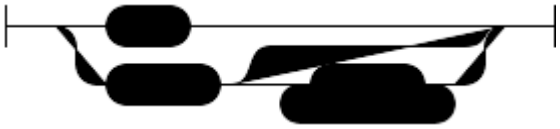
Yul identifiers consist of letters, dollar signs, underscores and numbers, but may not start with a number. In inline assembly there cannot be dots in user-defined identifiers. Instead see yulPath for expressions consisting of identifiers with dots.

yul-hex-number

Hex literals in Yul consist of a prefix and one or more hexadecimal digits.

yul-decimal-number

Decimal literals in Yul may be zero or any sequence of decimal digits without leading zeroes.



yul-string-literal

String literals in Yul consist of one or more double-quoted or single-quoted strings that may contain escape sequences and printable characters except unescaped line breaks or unescaped double-quotes or single-quotes, respectively.

pragma-token

Pragma token. Can contain any kind of symbol except a semicolon. Note that currently the solidity parser only allows a subset of this.

3.13 Using the Compiler

3.13.1 Using the Commandline Compiler

Note

This section does not apply to *solcjs*, not even if it is used in commandline mode.

Basic Usage

One of the build targets of the Solidity repository is `solc`, the Solidity commandline compiler. Using `solc --help` provides you with an explanation of all options. The compiler can produce various outputs, ranging from simple binaries and assembly over an abstract syntax tree (parse tree) to estimations of gas usage. If you only want to compile

a single file, you run it as `solc --bin sourceFile.sol` and it will print the binary. If you want to get some of the more advanced output variants of `solc`, it is probably better to tell it to output everything to separate files using `solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol`.

Optimizer Options

Before you deploy your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime (more specifically, it assumes each opcode is executed around 200 times). If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. If you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number. This parameter has effects on the following (this might change in the future):

- the size of the binary search in the function dispatch routine
- the way constants like large numbers or strings are stored

Base Path and Import Remapping

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide *path redirects* using `prefix=path` in the following way:

```
solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ file.sol
```

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`.

When accessing the filesystem to search for imports, *paths that do not start with ./ or ../* are treated as relative to the directories specified using `--base-path` and `--include-path` options (or the current working directory if base path is not specified). Furthermore, the part of the path added via these options will not appear in the contract metadata.

For security reasons the compiler has *restrictions on what directories it can access*. Directories of source files specified on the command-line and target paths of remappings are automatically allowed to be accessed by the file reader, but everything else is rejected by default. Additional paths (and their subdirectories) can be allowed via the `--allow-paths /sample/path,/another/sample/path` switch. Everything inside the path specified via `--base-path` is always allowed.

The above is only a simplification of how the compiler handles import paths. For a detailed explanation with examples and discussion of corner cases please refer to the section on *path resolution*.

Library Linking

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__$53aea86b7d70b31448b230b20ae141a537$__` (format was different <v0.5.0). These are placeholders for the actual library addresses. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name. The bytecode file will also contain lines of the form `// <placeholder> -> <fq library name>` at the end to help identify which libraries the placeholders represent. Note that the fully qualified library name is the path of its source file and the library name separated by `:`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points:

Either add `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD5678901234567890123456789012345678890"` to your command to provide an address for each library (use commas or spaces as separators) or store the string in a file (one library per line) and run `solc` using `--libraries fileName`.

Note

Starting Solidity 0.8.1 accepts `=` as separator between library and address, and `:` as a separator is deprecated. It will be removed in the future. Currently `--libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD5678901234567890123456789012345678890"` will work too.

If `solc` is called with the option `--standard-json`, it will expect a JSON input (as explained below) on the standard input, and return a JSON output on the standard output. This is the recommended interface for more complex and especially automated uses. The process will always terminate in a "success" state and report any errors via the JSON output. The option `--base-path` is also processed in standard-json mode.

If `solc` is called with the option `--link`, all input files are interpreted to be unlinked binaries (hex-encoded) in the `__$53aea86b7d70b31448b230b20ae141a537$__`-format given above and are linked in-place (if the input is read from stdin, it is written to stdout). All options except `--libraries` are ignored (including `-o`) in this case.

Warning

Manually linking libraries on the generated bytecode is discouraged because it does not update contract metadata. Since metadata contains a list of libraries specified at the time of compilation and bytecode contains a metadata hash, you will get different binaries, depending on when linking is performed.

You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

Note

The library placeholder used to be the fully qualified name of the library itself instead of the hash of it. This format is still supported by `solc --link` but the compiler will no longer output it. This change was made to reduce the likelihood of a collision between libraries, since only the first 36 characters of the fully qualified library name could be used.

3.13.2 Setting the EVM Version to Target

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviors.

Warning

Compiling for the wrong EVM version can result in wrong, strange and failing behavior. Please ensure, especially if running a private chain, that you use matching EVM versions.

On the command-line, you can select the EVM version as follows:

```
solc --evm-version <VERSION> contract.sol
```

In the *standard JSON interface*, use the `"evmVersion"` key in the `"settings"` field:

```
{
  "sources": {/* ... */},
  "settings": {
    "optimizer": {/* ... */},
    "evmVersion": "<VERSION>"
  }
}
```

Target Options

Below is a list of target EVM versions and the compiler-relevant changes introduced at each version. Backward compatibility is not guaranteed between each version.

- `homestead` (***support deprecated***)
  - (oldest version)
- `tangerineWhistle` (***support deprecated***)
  - Gas cost for access to other accounts increased, relevant for gas estimation and the optimizer.
  - All gas sent by default for external calls, previously a certain amount had to be retained.
- `spuriousDragon` (***support deprecated***)
  - Gas cost for the `exp` opcode increased, relevant for gas estimation and the optimizer.
- `byzantium` (***support deprecated***)
  - Opcodes `returndatacopy`, `returndatasize` and `staticcall` are available in assembly.
  - The `staticcall` opcode is used when calling non-library view or pure functions, which prevents the functions from modifying state at the EVM level, i.e., even applies when you use invalid type conversions.
  - It is possible to access dynamic data returned from function calls.
  - `revert` opcode introduced, which means that `revert()` will not waste gas.
- `constantinople`
  - Opcodes `create2`, `extcodehash`, `shl`, `shr` and `sar` are available in assembly.
  - Shifting operators use shifting opcodes and thus need less gas.
- `petersburg`
  - The compiler behaves the same way as with constantinople.
- `istanbul`
  - Opcodes `chainid` and `selfbalance` are available in assembly.
- `berlin`
  - Gas costs for `SLOAD`, `*CALL`, `BALANCE`, `EXT*` and `SELFDESTRUCT` increased. The compiler assumes cold gas costs for such operations. This is relevant for gas estimation and the optimizer.
- `london`
  - The block's base fee (EIP-3198 and EIP-1559) can be accessed via the global `block.basefee` or `basefee()` in inline assembly.
- `paris`

  - Introduces `prevrandao()` and `block.prevrandao`, and changes the semantics of the now deprecated `block.difficulty`, disallowing `difficulty()` in inline assembly (see EIP-4399).

- **shanghai**
  - Smaller code size and gas savings due to the introduction of `push0` (see EIP-3855).
- **cancun**
  - The block's blob base fee (EIP-7516 and EIP-4844) can be accessed via the global `block.blobbasefee` or `blobbasefee()` in inline assembly.
  - Introduces `blobhash()` in inline assembly and a corresponding global function to retrieve versioned hashes of blobs associated with the transaction (see EIP-4844).
  - Opcode `mcopy` is available in assembly (see EIP-5656).
  - Opcodes `tstore` and `tload` are available in assembly (see EIP-1153).
- `prague`
- **osaka (default)**
  - `clz` builtin function is available in inline assembly. (EIP-7939)
- `amsterdam` (**experimental**)

3.13.3 Compiler Input and Output JSON Description

The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The same interface is provided by all distributions of the compiler.

The fields are generally subject to change, some are optional (as noted), but we try to only make backwards compatible changes.

The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The standard error output is not used and the process will always terminate in a "success" state, even if there were errors. Errors are always reported as part of the JSON output.

The following subsections describe the format through an example. Comments are of course not permitted and used here only for explanatory purposes.

Input Description

```
{
  // Required: Source code language. Currently supported are "Solidity", "Yul",
↪"SolidityAST" (experimental), "EVMAssembly" (experimental).
  "language": "Solidity",
  // Required
  "sources":
  {
    // The keys here are the "global" names of the source files,
    // imports can use other files via remappings (see below).
    "myFile.sol":
    {
      // Optional: keccak256 hash of the source file
      // It is used to verify the retrieved content if imported via URLs.
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): URL(s) to the source file.
      // URL(s) should be imported in this order and the result checked against the
```

(continues on next page)

(continued from previous page)

```
      // keccak256 hash (if available). If the hash doesn't match or none of the
      // URL(s) result in success, an error should be raised.
      // Using the commandline interface only filesystem paths are supported.
      // With the JavaScript interface the URL will be passed to the user-supplied
      // read callback, so any URL supported by the callback can be used.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // If files are used, their directories should be added to the command-line via
        // `--allow-paths <path>`.
      ]
    },
    "settable":
    {
      // Optional: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "urls" is used): literal contents of the source file
      "content": "contract settable is owned { uint256 private x = 0; function␣
↪set(uint256 _x) public { if (msg.sender == owner) x = _x; } }"
    },
    "myFile.sol_json.ast":
    {
      // If language is set to "SolidityAST", an AST needs to be supplied under the "ast
↪" key
      // and there can be only one source file present.
      // The format is the same as used by the `ast` output.
      // Note that importing ASTs is experimental and in particular that:
      // - importing invalid ASTs can produce undefined results and
      // - no proper error reporting is available on invalid ASTs.
      // Furthermore, note that the AST import only consumes the fields of the AST as
      // produced by the compiler in "stopAfter": "parsing" mode and then re-performs
      // analysis, so any analysis-based annotations of the AST are ignored upon import.
      "ast": { ... }
    },
    "myFile_evm.json":
    {
      // If language is set to "EVMAssembly", an EVM Assembly JSON object needs to be␣
↪supplied
      // under the "assemblyJson" key and there can be only one source file present.
      // The format is the same as used by the `evm.legacyAssembly` output or `--asm-json`
      // output on the command line.
      // Note that importing EVM assembly is experimental.
      "assemblyJson":
      {
        ".code": [ ... ],
        ".data": { ... }, // optional
        "sourceList": [ ... ] // optional (if no `source` node was defined in any `.code`␣
↪object)
      }
    }
```

(continues on next page)

(continued from previous page)

```
  },
  // Optional
  "settings":
  {
    // Optional: Stop compilation after the given stage. Currently only "parsing" is␣
↪valid here
    "stopAfter": "parsing",
    // Optional: List of remappings
    "remappings": [ ":g=/dir" ],
    // Optional: Experimental mode toggle (Default: false)
    // Makes it possible to use experimental features (but does not enable any such␣
↪feature by itself).
    // The use of this mode is recorded in contract metadata.
    "experimental": true,
    // Optional: Optimizer settings
    "optimizer": {
      // Turn on the optimizer. Optional. Default: false.
      // NOTE: The state of the optimizer is fully determined by the 'details' dict and␣
↪this setting
      // only affects its defaults - when enabled, all components default to being␣
↪enabled.
      // The opposite is not true - there are several components that always default to␣
↪being
      // enabled an can only be explicitly disabled via 'details'.
      // WARNING: Before version 0.8.6 omitting this setting was not equivalent to␣
↪setting
      // it to false and would result in all components being disabled instead.
      // WARNING: Enabling optimizations for EVMAssembly input is allowed but not␣
↪necessary under normal
      // circumstances. It forces the opcode-based optimizer to run again and can␣
↪produce bytecode that
      // is not reproducible from metadata.
      "enabled": true,
      // Optimize for how many times you intend to run the code. Optional. Default: 200.
      // Lower values will optimize more for initial deployment cost, higher
      // values will optimize more for high-frequency usage.
      "runs": 200,
      // State of all optimizer components. Optional.
      // Default values are determined by whether the optimizer is enabled or not.
      // Note that the 'enabled' setting only affects the defaults here and has no effect␣
↪when
      // all values are provided explicitly.
      "details": {
        // Peephole optimizer (opcode-based). Optional. Default: true.
        // Default for EVMAssembly input: false when optimization is not enabled.
        // NOTE: Always runs (even with optimization disabled) except for EVMAssembly␣
↪input or when explicitly turned off here.
        "peephole": true,
        // Inliner (opcode-based). Optional. Default: true when optimization is enabled.
        "inliner": false,
        // Unused JUMPDEST remover (opcode-based). Optional. Default: true.
        // Default for EVMAssembly input: false when optimization is not enabled.
```

(continues on next page)

(continued from previous page)

```
      // NOTE: Always runs (even with optimization disabled) except for EVMAssembly␣
↪input or when explicitly turned off here.
      "jumpdestRemover": true,
      // Literal reordering (codegen-based). Optional. Default: true when optimization␣
↪is enabled.
      // Moves literals to the right of commutative binary operators during code␣
↪generation, helping exploit associativity.
      "orderLiterals": false,
      // Block deduplicator (opcode-based). Optional. Default: true when optimization␣
↪is enabled.
      // Unifies assembly code blocks that share content.
      "deduplicate": false,
      // Common subexpression elimination (opcode-based). Optional. Default: true when␣
↪optimization is enabled.
      // This is the most complicated step but can also provide the largest gain.
      "cse": false,
      // Constant optimizer (opcode-based). Optional. Default: true when optimization␣
↪is enabled.
      // Tries to find better representations of literal numbers and strings, that␣
↪satisfy the
      // size/cost trade-off determined by the 'runs' setting.
      "constantOptimizer": false,
      // Unchecked loop increment (codegen-based). Optional. Default: true.
      // Use unchecked arithmetic when incrementing the counter of 'for' loops under␣
↪certain circumstances.
      // NOTE: Always runs (even with optimization disabled) unless explicitly turned␣
↪off here.
      "simpleCounterForLoopUncheckedIncrement": true,
      // Yul optimizer. Optional. Default: true when optimization is enabled.
      // Used to optimize the IR produced by the Yul IR-based pipeline as well as␣
↪inline assembly
      // and utility Yul code generated by the compiler.
      // NOTE: Before Solidity 0.6.0 the default was false.
      "yul": false,
      // Tuning options for the Yul optimizer. Optional.
      "yulDetails": {
        // Improve allocation of stack slots for variables, can free up stack slots␣
↪early.
        // Optional. Default: true if Yul optimizer is enabled.
        "stackAllocation": true,
        // Optimization step sequence.
        // The general form of the value is "<main sequence>:<cleanup sequence>".
        // The setting is optional and when omitted, default values are used for both␣
↪sequences.
        // If the value does not contain the ':' delimiter, it is interpreted as the␣
↪main
        // sequence and the default is used for the cleanup sequence.
        // To make one of the sequences empty, the delimiter must be present at the␣
↪first or last position.
        // In particular if the whole value consists only of the delimiter, both␣
↪sequences are empty.
        // Note that there are several hard-coded steps that always run, even when␣
```

(continues on next page)

(continued from previous page)

```
↪both sequences are empty.
          // For more information see "The Optimizer > Selecting Optimizations".
          "optimizerSteps": "dfDvulfnTUtnIf..."
        }
      }
    },
    // Version of the EVM to compile for (optional).
    // Affects type checking and code generation. Can be homestead,
    // tangerineWhistle, spuriousDragon, byzantium, constantinople,
    // petersburg, istanbul, berlin, london, paris, shanghai, cancun,
    // prague, osaka (default), amsterdam (experimental), or @future (experimental).
    "evmVersion": "osaka",
    // Optional: Change compilation pipeline to go through the Yul intermediate␣
↪representation.
    // This is false by default.
    "viaIR": true,
    // Optional: Turn on SSA CFG-based code generation via the IR (experimental).
    // Implies viaIR: true. This is false by default.
    "viaSSACFG": false,
    // Optional: Debugging settings
    "debug": {
      // How to treat revert (and require) reason strings. Settings are
      // "default", "strip", "debug" and "verboseDebug".
      // "default" does not inject compiler-generated revert strings and keeps user-
↪supplied ones.
      // "strip" removes all revert strings (if possible, i.e. if literals are used)␣
↪keeping side-effects.
      // NOTE: "strip" does not remove custom errors.
      // "debug" injects strings for compiler-generated internal reverts, implemented␣
↪for ABI encoders V1 and V2 for now.
      // "verboseDebug" even appends further information to user-supplied revert strings␣
↪(not yet implemented)
      "revertStrings": "default",
      // Optional: How much extra debug information to include in comments in the␣
↪produced EVM
      // assembly and Yul code. Available components are:
      // - `location`: Annotations of the form `@src <index>:<start>:<end>` indicating the
      //    location of the corresponding element in the original Solidity file, where:
      //     - `<index>` is the file index matching the `@use-src` annotation,
      //     - `<start>` is the index of the first byte at that location,
      //     - `<end>` is the index of the first byte after that location.
      // - `snippet`: A single-line code snippet from the location indicated by `@src`.
      //     The snippet is quoted and follows the corresponding `@src` annotation.
      // - `ast-id`: Annotations of the form `@ast-id <id>` over elements that can be␣
↪mapped back to a definition in the original Solidity file.
      //   `<id>` is a node ID in the Solidity AST ('ast' output).
      // - `ethdebug`: Ethdebug annotations (experimental). Automatically enabled when␣
↪any ethdebug output is requested.
      // - `*`: Wildcard value that can be used to request all non-experimental␣
↪components.
      "debugInfo": ["location", "snippet", "ast-id", "ethdebug"]
    },
```

(continues on next page)

(continued from previous page)

```
    // Metadata settings (optional)
    "metadata": {
      // The CBOR metadata is appended at the end of the bytecode by default.
      // Setting this to false omits the metadata from the runtime and deploy time code.
      "appendCBOR": true,
      // Use only literal content and not URLs (false by default)
      "useLiteralContent": true,
      // Use the given hash method for the metadata hash that is appended to the␣
↪bytecode.
      // The metadata hash can be removed from the bytecode via option "none".
      // The other options are "ipfs" and "bzzr1".
      // If the option is omitted, "ipfs" is used by default.
      "bytecodeHash": "ipfs"
    },
    // Addresses of the libraries. If not all libraries are given here,
    // it can result in unlinked objects whose output data is different.
    "libraries": {
      // The top level key is the name of the source file where the library is used.
      // If remappings are used, this source file should match the global path
      // after remappings were applied.
      // If this key is an empty string, that refers to a global level.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    },
    // The following can be used to select desired outputs based
    // on file and contract names.
    // If this field is omitted, then the compiler loads and does type checking,
    // but will not generate any outputs apart from errors.
    // The first level key is the file name and the second level key is the contract␣
↪name.
    // An empty contract name is used for outputs that are not tied to a contract
    // but to the whole source file like the AST.
    // A star as contract name refers to all contracts in the file.
    // Similarly, a star as a file name matches all files.
    // To select all outputs the compiler can possibly generate, with the exclusion of
    // Yul intermediate representation outputs, use
    // "outputSelection: { "*": { "*": [ "*" ], "": [ "*" ] } }"
    // but note that this might slow down the compilation process needlessly.
    //
    // The available output types are as follows:
    //
    // File level (needs empty string as contract name):
    //   ast - AST of all source files
    //
    // Contract level (needs the contract name or "*"):
    //   abi - ABI
    //   devdoc - Developer documentation (natspec)
    //   userdoc - User documentation (natspec)
    //   metadata - Metadata
    //   ir - Yul intermediate representation of the code before optimization
    //   irAst - AST of Yul intermediate representation of the code before optimization␣
```

(continues on next page)

(continued from previous page)

```
↪(experimental)
    //   irOptimized - Intermediate representation after optimization
    //   irOptimizedAst - AST of intermediate representation after optimization␣
↪(experimental)
    //   storageLayout - Slots, offsets and types of the contract's state variables in␣
↪storage
    //   transientStorageLayout - Slots, offsets and types of the contract's state␣
↪variables in transient storage
    //   evm.assembly - New assembly format
    //   evm.legacyAssembly - Old-style assembly format in JSON
    //   evm.bytecode.ethdebug - Debug information in ethdebug format (ethdebug/format/
↪program schema for creation bytecode). Can only be requested when compiling via IR.␣
↪(experimental)
    //   evm.deployedBytecode.ethdebug - Debug information in ethdebug format (ethdebug/
↪format/program schema for deployed bytecode). Can only be requested when compiling via␣
↪IR. (experimental)
    //   evm.bytecode.functionDebugData - Debugging information at function level
    //   evm.bytecode.object - Bytecode object
    //   evm.bytecode.opcodes - Opcodes list
    //   evm.bytecode.sourceMap - Source mapping (useful for debugging)
    //   evm.bytecode.linkReferences - Link references (if unlinked object)
    //   evm.bytecode.generatedSources - Sources generated by the compiler
    //   evm.deployedBytecode* - Deployed bytecode (has all the options that evm.
↪bytecode has)
    //   evm.deployedBytecode.immutableReferences - Map from AST ids to bytecode ranges␣
↪that reference immutables
    //   evm.methodIdentifiers - The list of function hashes
    //   evm.gasEstimates - Function gas estimates
    //   yulCFGJson - Control Flow Graph (CFG) of the Single Static Assignment (SSA)␣
↪form of the contract (experimental)
    //
    // Global level (needs "*" as file name and "*" as contract name):
    //   ethdebug.resources - Global ethdebug output (ethdebug/format/info/resources␣
↪schema) containing source list and compiler info (experimental)
    //   ethdebug.compilation - Global ethdebug compilation output (the 'compilation' key␣
↪from ethdebug/format/info/resources schema) (experimental)
    //
    // Note that using `evm`, `evm.bytecode`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to request␣
↪everything.
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // Enable the metadata and bytecode outputs of␣
↪every single contract.
          , "evm.bytecode.sourceMap" // Enable the source map output of every single␣
↪contract.
        ],
        "": [
          "ast" // Enable the AST output of every single file.
        ]
```

(continues on next page)

(continued from previous page)

```
      },
      // Enable the abi and opcodes output of MyContract defined in file def.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    },
    // The modelChecker object is experimental and subject to changes.
    "modelChecker":
    {
      // Chose which contracts should be analyzed as the deployed one.
      "contracts":
      {
        "source1.sol": ["contract1"],
        "source2.sol": ["contract2", "contract3"]
      },
      // Choose how division and modulo operations should be encoded.
      // When using `false` they are replaced by multiplication with slack
      // variables. This is the default.
      // Using `true` here is recommended if you are using the CHC engine
      // and not using Spacer as the Horn solver (using Eldarica, for example).
      // See the Formal Verification section for a more detailed explanation of this␣
↪option.
      "divModNoSlacks": false,
      // Choose which model checker engine to use: all (default), bmc, chc, none.
      "engine": "chc",
      // Choose whether external calls should be considered trusted in case the
      // code of the called function is available at compile-time.
      // For details see the SMTChecker section.
      "extCalls": "trusted",
      // Choose which types of invariants should be reported to the user: contract,␣
↪reentrancy.
      "invariants": ["contract", "reentrancy"],
      // Choose whether to output all proved targets. The default is `false`.
      "showProvedSafe": true,
      // Choose whether to output all unproved targets. The default is `false`.
      "showUnproved": true,
      // Choose whether to output all unsupported language features. The default is␣
↪`false`.
      "showUnsupported": true,
      // Choose which solvers should be used, if available.
      // See the Formal Verification section for the solvers description.
      "solvers": ["cvc5", "smtlib2", "z3"],
      // Choose which targets should be checked: constantCondition,
      // underflow, overflow, divByZero, balance, assert, popEmptyArray, outOfBounds.
      // If the option is not given all targets are checked by default,
      // except underflow/overflow for Solidity >=0.8.7.
      // See the Formal Verification section for the targets description.
      "targets": ["underflow", "overflow", "assert"],
      // Timeout for each SMT query in milliseconds.
      // If this option is not given, the SMTChecker will use a deterministic
      // resource limit by default.
      // A given timeout of 0 means no resource/time restrictions for any query.
```

(continues on next page)

(continued from previous page)

```
      "timeout": 20000
    }
  }
}
```

Output Description

```
{
  // Optional: not present if no errors/warnings/infos were encountered
  "errors": [
    {
      // Optional: Location within the source file.
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      },
      // Optional: Further locations (e.g. places of conflicting declarations)
      "secondarySourceLocations": [
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "Other declaration is here:"
        }
      ],
      // Mandatory: Error type, such as "TypeError", "InternalCompilerError", "Exception
↪", etc.
      // See below for complete list of types.
      "type": "TypeError",
      // Mandatory: Component where the error originated, such as "general" etc.
      "component": "general",
      // Mandatory ("error", "warning" or "info", but please note that this may be␣
↪extended in the future)
      "severity": "error",
      // Optional: unique code for the cause of the error
      "errorCode": "3141",
      // Mandatory
      "message": "Invalid keyword",
      // Optional: the message formatted with source location
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // This contains the file-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "sources": {
    "sourceFile.sol": {
      // Identifier of the source (used in source maps)
      "id": 1,
      // The AST object
      "ast": {}
```

(continues on next page)

(continued from previous page)

```
    }
  },
  // This contains the contract-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "contracts": {
    "sourceFile.sol": {
      // If the language used has no contract names, this field should equal to an empty␣
↪string.
      "ContractName": {
        // The Ethereum Contract ABI. If empty, it is represented as an empty array.
        // See https://docs.soliditylang.org/en/develop/abi-spec.html
        "abi": [],
        // See the Metadata Output documentation (serialised JSON string)
        "metadata": "{/* ... */}",
        // User documentation (natspec)
        "userdoc": {},
        // Developer documentation (natspec)
        "devdoc": {},
        // Intermediate representation before optimization (string)
        "ir": "",
        // AST of intermediate representation before optimization
        "irAst":  {/* ... */},
        // Intermediate representation after optimization (string)
        "irOptimized": "",
        // AST of intermediate representation after optimization
        "irOptimizedAst": {/* ... */},
        // See the Storage Layout documentation.
        "storageLayout": {"storage": [/* ... */], "types": {/* ... */} },
        // See the Storage Layout documentation.
        "transientStorageLayout": {"storage": [/* ... */], "types": {/* ... */} },
        // EVM-related outputs
        "evm": {
          // Assembly (string)
          "assembly": "",
          // Old-style assembly (object)
          "legacyAssembly": {},
          // Bytecode and related details.
          "bytecode": {
            // Ethdebug output (experimental)
            "ethdebug": {/* ... */},
            // Debugging data at the level of functions.
            "functionDebugData": {
              // Now follows a set of functions including compiler-internal and
              // user-defined function. The set does not have to be complete.
              "@mint_13": { // Internal name of the function
                "entryPoint": 128, // Byte offset into the bytecode where the function␣
↪starts (optional)
                "id": 13, // AST ID of the function definition or null for compiler-
↪internal functions (optional)
                "parameterSlots": 2, // Number of EVM stack slots for the function␣
↪parameters (optional)
                "returnSlots": 1 // Number of EVM stack slots for the return values␣
```

(continues on next page)

(continued from previous page)

```
↪(optional)
            }
          },
          // The bytecode as a hex string.
          "object": "00fe",
          // Opcodes list (string)
          "opcodes": "",
          // The source mapping as a string. See the source mapping definition.
          "sourceMap": "",
          // Array of sources generated by the compiler. Currently only
          // contains a single Yul file.
          "generatedSources": [{
            // Yul AST
            "ast": {/* ... */},
            // Source file in its text form (may contain comments)
            "contents":"{ function abi_decode(start, end) -> data { data :=␣
↪calldataload(start) } }",
            // Source file ID, used for source references, same "namespace" as the␣
↪Solidity source files
            "id": 2,
            "language": "Yul",
            "name": "#utility.yul"
          }],
          // If given, this is an unlinked object.
          "linkReferences": {
            "libraryFile.sol": {
              // Byte offsets into the bytecode.
              // Linking replaces the 20 bytes located there.
              "Library1": [
                { "start": 0, "length": 20 },
                { "start": 200, "length": 20 }
              ]
            }
          }
        },
        "deployedBytecode": {
          // Ethdebug output (experimental)
          "ethdebug": {/* ... */},
          /* ..., */ // The same layout as above.
          "immutableReferences": {
            // There are two references to the immutable with AST ID 3, both 32 bytes␣
↪long. One is
            // at bytecode offset 42, the other at bytecode offset 80.
            "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
          }
        },
        // The list of function hashes
        "methodIdentifiers": {
          "delegate(address)": "5c19a95c"
        },
        // Function gas estimates
        "gasEstimates": {
```

(continues on next page)

(continued from previous page)

```
            "creation": {
              "codeDepositCost": "420000",
              "executionCost": "infinite",
              "totalCost": "infinite"
            },
            "external": {
              "delegate(address)": "25000"
            },
            "internal": {
              "heavyLifting()": "infinite"
            }
          },
          // Yul CFG representation of the SSA form (experimental)
          "yulCFGJson": {/* ... */}
        }
      }
    }
  },
  // Global Ethdebug output (experimental)
  "ethdebug": {
    // Requested via ethdebug.resources output selection
    "resources": {/* ... */},
    // Requested via ethdebug.compilation output selection
    "compilation": {/* ... */}
  }
}
```

Error Types

1. `JSONError`: JSON input doesn't conform to the required format, e.g. input is not a JSON object, the language is not supported, etc.
2. `IOError`: IO and import processing errors, such as unresolvable URL or hash mismatch in supplied sources.
3. `ParserError`: Source code doesn't conform to the language rules.
4. `DocstringParsingError`: The NatSpec tags in the comment block cannot be parsed.
5. `SyntaxError`: Syntactical error, such as `continue` is used outside of a `for` loop.
6. `DeclarationError`: Invalid, unresolvable or clashing identifier names. e.g. `Identifier not found`
7. `TypeError`: Error within the type system, such as invalid type conversions, invalid assignments, etc.
8. `UnimplementedFeatureError`: Feature is not supported by the compiler, but is expected to be supported in future versions.
9. `InternalCompilerError`: Internal bug triggered in the compiler - this should be reported as an issue.
10. `Exception`: Unknown failure during compilation - this should be reported as an issue.
11. `CompilerError`: Invalid use of the compiler stack - this should be reported as an issue.
12. `FatalError`: Fatal error not processed correctly - this should be reported as an issue.
13. `YulException`: Error during Yul code generation - this should be reported as an issue.
14. `Warning`: A warning, which didn't stop the compilation, but should be addressed if possible.

15. `Info`: Information that the compiler thinks the user might find useful, but is not dangerous and does not necessarily need to be addressed.

3.13.4 Experimental Mode

Some language and compiler features included in stable releases are not themselves considered stable. They are sparsely documented, if at all, often not adequately tested, and thus not yet intended for production use. In many cases it is possible to develop a big feature incrementally, with each iteration being already stable. Sometimes, however, it is preferable to start with a prototype and stabilize it over multiple releases, while receiving feedback from users. To prevent accidental use, such features can be only accessed by enabling the experimental mode.

There are no backwards compatibility guarantees for experimental features. They are subject to change in breaking ways in non-breaking releases of the compiler. Only major changes affecting them are recorded in the changelog.

To enable the experimental mode, use the `--experimental` flag on the command line, or the analogous `settings.experimental` boolean setting in the Standard JSON input.

Note that the use of this mode is recorded in the metadata:

- `experimental` flag in CBOR metadata is set to `true`,
- `settings.experimental` in JSON metadata is set to `true`,

Note

Prior to version 0.8.35, most of the experimental features were usable without any extra safeguards. Some were gated behind `pragma experimental`, but this was not done consistently. The information about them was also only recorded in CBOR metadata and even then not always. The main goal of the experimental mode is to systematize this and make users fully aware when relying on features which are unfinished or not production-ready.

The table below details all currently available experimental features.

Feature	ID	Affects bytecode	Flag/pragma
AST import	`ast-impor`	yes	`--import-ast`
LSP	`lsp`	no	`--lsp`
EVM Assembly import	`evmasm-im`	yes	`--import-asm-json`
Generic Solidity	`generic-s`	yes	`pragma experimental solidity`
IR AST	`ir-ast`	no	`--ir-ast-json`, `--ir-optimized-ast-json`
Non-mainnet EVMs	`evm`	yes	`--evm-version <version name>`
Ethdebug	`ethdebug`	no	`--ethdebug-resources`, `--ethdebug-compilation`, `--ethdebug-program`, `--ethdebug-program-runtime`, `--debug-info ethdebug`
SSA CFG	`ssa-cfg`	no	`--yul-cfg-json`
		yes	`--via-ssa-cfg`

3.14 Analysing the Compiler Output

It is often useful to look at the assembly code generated by the compiler. The generated binary, i.e., the output of `solc --bin contract.sol`, is generally difficult to read. It is recommended to use the flag `--asm` to analyse the assembly output. Even for large contracts, looking at a visual diff of the assembly before and after a change is often very enlightening.

Consider the following contract (named, say `contract.sol`):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

The following would be the output of `solc --asm contract.sol`

```
======= contract.sol:C =======
EVM assembly:
    /* "contract.sol":0:86  contract C {... */
  mstore(0x40, 0x80)
  callvalue
  dup1
  iszero
  tag_1
  jumpi
  0x00
  dup1
  revert
tag_1:
  pop
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x00
  codecopy
  0x00
  return
stop

sub_0: assembly {
        /* "contract.sol":0:86  contract C {... */
      mstore(0x40, 0x80)
      callvalue
      dup1
      iszero
      tag_1
      jumpi
      0x00
      dup1
      revert
    tag_1:
```

(continues on next page)

(continued from previous page)

```
    pop
    jumpi(tag_2, lt(calldatasize, 0x04))
    shr(0xe0, calldataload(0x00))
    dup1
    0x901717d1
    eq
    tag_3
    jumpi
  tag_2:
    0x00
    dup1
    revert
      /* "contract.sol":17:84  function one() public pure returns (uint) {... */
  tag_3:
    tag_4
    tag_5
    jump  // in
  tag_4:
    mload(0x40)
    tag_6
    swap2
    swap1
    tag_7
    jump  // in
  tag_6:
    mload(0x40)
    dup1
    swap2
    sub
    swap1
    return
  tag_5:
      /* "contract.sol":53:57  uint */
    0x00
      /* "contract.sol":76:77  1 */
    0x01
      /* "contract.sol":69:77  return 1 */
    swap1
    pop
      /* "contract.sol":17:84  function one() public pure returns (uint) {... */
    swap1
    jump  // out
      /* "#utility.yul":7:125   */
  tag_10:
      /* "#utility.yul":94:118   */
    tag_12
      /* "#utility.yul":112:117   */
    dup2
      /* "#utility.yul":94:118   */
    tag_13
    jump  // in
  tag_12:
```

(continues on next page)

(continued from previous page)

```
      /* "#utility.yul":89:92   */
    dup3
      /* "#utility.yul":82:119   */
    mstore
      /* "#utility.yul":72:125   */
    pop
    pop
    jump  // out
      /* "#utility.yul":131:353   */
  tag_7:
    0x00
      /* "#utility.yul":262:264   */
    0x20
      /* "#utility.yul":251:260   */
    dup3
      /* "#utility.yul":247:265   */
    add
      /* "#utility.yul":239:265   */
    swap1
    pop
      /* "#utility.yul":275:346   */
    tag_15
      /* "#utility.yul":343:344   */
    0x00
      /* "#utility.yul":332:341   */
    dup4
      /* "#utility.yul":328:345   */
    add
      /* "#utility.yul":319:325   */
    dup5
      /* "#utility.yul":275:346   */
    tag_10
    jump  // in
  tag_15:
      /* "#utility.yul":229:353   */
    swap3
    swap2
    pop
    pop
    jump  // out
      /* "#utility.yul":359:436   */
  tag_13:
    0x00
      /* "#utility.yul":425:430   */
    dup2
      /* "#utility.yul":414:430   */
    swap1
    pop
      /* "#utility.yul":404:436   */
    swap2
    swap1
    pop
```

(continues on next page)

(continued from previous page)

```
      jump  // out

    auxdata:␣
↪0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c6378273
}
```

Alternatively, the above output can also be obtained from Remix, under the option "Compilation Details" after compiling a contract.

Notice that the `asm` output starts with the creation / constructor code. The deploy code is provided as part of the sub object (in the above example, it is part of the sub-object `sub_0`). The `auxdata` field corresponds to the contract *metadata*. The comments in the assembly output point to the source location. Note that `#utility.yul` is an internally generated file of utility functions that can be obtained using the flags `--combined-json generated-sources, generated-sources-runtime`.

Similarly, the optimized assembly can be obtained with the command: `solc --optimize --asm contract.sol`. Often times, it is interesting to see if two different sources in Solidity result in the same optimized code. For example, to see if the expressions `(a * b) / c`, `a * b / c` generates the same bytecode. This can be easily done by taking a `diff` of the corresponding assembly output, after potentially stripping comments that reference the source locations.

Note

The `--asm` output is not designed to be machine readable. Therefore, there may be breaking changes on the output between minor versions of solc.

3.15 Solidity IR-based Codegen Changes

Solidity can generate EVM bytecode in two different ways: Either directly from Solidity to EVM opcodes ("old codegen") or through an intermediate representation ("IR") in Yul ("new codegen" or "IR-based codegen").

The IR-based code generator was introduced with an aim to not only allow code generation to be more transparent and auditable but also to enable more powerful optimization passes that span across functions.

You can enable it on the command-line using `--via-ir` or with the option `{"viaIR": true}` in standard-json and we encourage everyone to try it out!

For several reasons, there are tiny semantic differences between the old and the IR-based code generator, mostly in areas where we would not expect people to rely on this behavior anyway. This section highlights the main differences between the old and the IR-based codegen.

3.15.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- The order of state variable initialization has changed in case of inheritance.

  The order used to be:

  - All state variables are zero-initialized at the beginning.
  - Evaluate base constructor arguments from most derived to most base contract.
  - Initialize all state variables in the whole inheritance hierarchy from most base to most derived.
  - Run the constructor, if present, for all contracts in the linearized hierarchy from most base to most derived.

  New order:

  - All state variables are zero-initialized at the beginning.
  - Evaluate base constructor arguments from most derived to most base contract.
  - For every contract in order from most base to most derived in the linearized hierarchy:
    1. Initialize state variables.
    2. Run the constructor (if present).

This causes differences in contracts where the initial value of a state variable relies on the result of the constructor in another contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract A {
    uint x;
    constructor() {
        x = 42;
    }
    function f() public view returns(uint256) {
        return x;
    }
}
contract B is A {
    uint public y = f();
}
```

Previously, `y` would be set to 0. This is due to the fact that we would first initialize state variables: First, `x` is set to 0, and when initializing `y`, `f()` would return 0 causing `y` to be 0 as well. With the new rules, `y` will be set to 42. We first initialize `x` to 0, then call A's constructor which sets `x` to 42. Finally, when initializing `y`, `f()` returns 42 causing `y` to be 42.

- When storage structs are deleted, every storage slot that contains a member of the struct is set to zero entirely. Formerly, padding space was left untouched. Consequently, if the padding space within a struct is used to store data (e.g. in the context of a contract upgrade), you have to be aware that `delete` will now also clear the added member (while it wouldn't have been cleared in the past).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract C {
    struct S {
        uint64 y;
        uint64 z;
    }
    S s;
    function f() public {
        // ...
        delete s;
        // s occupies only first 16 bytes of the 32 bytes slot
        // delete will write zero to the full slot
    }
}
```

We have the same behavior for implicit delete, for example when array of structs is shortened.

- Function modifiers are implemented in a slightly different way regarding function parameters and return variables. This especially has an effect if the placeholder `_;` is evaluated multiple times in a modifier. In the old code generator, each function parameter and return variable has a fixed slot on the stack. If the function is run multiple times because `_;` is used multiple times or used in a loop, then a change to the function parameter's or return variable's value is visible in the next execution of the function. The new code generator implements modifiers using actual functions and passes function parameters on. This means that multiple evaluations of a function's body will get the same values for the parameters, and the effect on return variables is that they are reset to their default (zero) value for each execution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0;
contract C {
    function f(uint a) public pure mod() returns (uint r) {
        r = a++;
    }
    modifier mod() { _; _; }
}
```

If you execute `f(0)` in the old code generator, it will return `1`, while it will return `0` when using the new code generator.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract C {
    bool active = true;
    modifier mod()
    {
        _;
        active = false;
        _;
    }
    function foo() external mod() returns (uint ret)
    {
        if (active)
            ret = 1; // Same as ``return 1``
    }
}
```

The function `C.foo()` returns the following values:

  - Old code generator: `1` as the return variable is initialized to `0` only once before the first `_;` evaluation and then overwritten by the `return 1;`. It is not initialized again for the second `_;` evaluation and `foo()` does not explicitly assign it either (due to `active == false`), thus it keeps its first value.
  - New code generator: `0` as all parameters, including return parameters, will be re-initialized before each `_;` evaluation.

- For the old code generator, the evaluation order of expressions is unspecified. For the new code generator, we try to evaluate in source order (left to right), but do not guarantee it. This can lead to semantic differences.

  For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
```

(continues on next page)

(continued from previous page)

```
    function preincr_u8(uint8 a) public pure returns (uint8) {
        return ++a + a;
    }
}
```

The function `preincr_u8(1)` returns the following values:

- Old code generator: `3` (`1 + 2`) but the return value is unspecified in general
- New code generator: `4` (`2 + 2`) but the return value is not guaranteed

On the other hand, function argument expressions are evaluated in the same order by both code generators with the exception of the global functions `addmod` and `mulmod`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function add(uint8 a, uint8 b) public pure returns (uint8) {
        return a + b;
    }
    function g(uint8 a, uint8 b) public pure returns (uint8) {
        return add(++a + ++b, a + b);
    }
}
```

The function `g(1, 2)` returns the following values:

- Old code generator: `10` (`add(2 + 3, 2 + 3)`) but the return value is unspecified in general
- New code generator: `10` but the return value is not guaranteed

The arguments to the global functions `addmod` and `mulmod` are evaluated right-to-left by the old code generator and left-to-right by the new code generator. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f() public pure returns (uint256 aMod, uint256 mMod) {
        uint256 x = 3;
        // Old code gen: add/mulmod(5, 4, 3)
        // New code gen: add/mulmod(4, 5, 5)
        aMod = addmod(++x, ++x, x);
        mMod = mulmod(++x, ++x, x);
    }
}
```

The function `f()` returns the following values:

- Old code generator: `aMod = 0` and `mMod = 2`
- New code generator: `aMod = 4` and `mMod = 0`

- The new code generator imposes a hard limit of `type(uint64).max` (`0xffffffffffffffff`) for the free memory pointer. Allocations that would increase its value beyond this limit revert. The old code generator does not have this limit.

  For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
contract C {
    function f() public {
        uint[] memory arr;
        // allocation size: 576460752303423481
        // assumes freeMemPtr points to 0x80 initially
        uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 -␣
↪31) / 32;
        // freeMemPtr overflows UINT64_MAX
        arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);
    }
}
```

The function `f()` behaves as follows:

- Old code generator: runs out of gas while zeroing the array contents after the large memory allocation
- New code generator: reverts due to free memory pointer overflow (does not run out of gas)

3.15.2 Internals

Internal function pointers

The old code generator uses code offsets or tags for values of internal function pointers. This is especially complicated since these offsets are different at construction time and after deployment and the values can cross this border via storage. Because of that, both offsets are encoded at construction time into the same value (into different bytes).

In the new code generator, function pointers use internal IDs that are allocated in sequence. Since calls via jumps are not possible, calls through function pointers always have to use an internal dispatch function that uses the `switch` statement to select the right function.

The ID `0` is reserved for uninitialized function pointers which then cause a panic in the dispatch function when called.

In the old code generator, internal function pointers are initialized with a special function that always causes a panic. This causes a storage write at construction time for internal function pointers in storage.

Note

The compiler is free to omit internal functions that are never explicitly referenced by name. As a consequence, assigning to a function type variable in inline assembly does not guarantee that the assigned value will be included in the internal dispatch. The function must also be explicitly referenced elsewhere in the code.

Cleanup

The old code generator only performs cleanup before an operation whose result could be affected by the values of the dirty bits. The new code generator performs cleanup after any operation that can result in dirty bits. The hope is that the optimizer will be powerful enough to eliminate redundant cleanup operations.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f(uint8 a) public pure returns (uint r1, uint r2)
```

(continues on next page)

(continued from previous page)

```
    {
        a = ~a;
        assembly {
            r1 := a
        }
        r2 = a;
    }
}
```

The function `f(1)` returns the following values:

- Old code generator: (`fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe`, `0000000000000000000000000000000000000000000000000000000000000001`)
- New code generator: (`0000000000000000000000000000000000000000000000000000000000000001`, `0000000000000000000000000000000000000000000000000000000000000001`)

Note that, unlike the new code generator, the old code generator does not perform a cleanup after the bit-not assignment (`a = ~a`). This results in different values being assigned (within the inline assembly block) to return value `r1` between the old and new code generators. However, both code generators perform a cleanup before the new value of `a` is assigned to `r2`.

3.16 Layout of State Variables in Storage and Transient Storage

Note

The rules described in this section apply for both storage and transient storage data locations. The layouts are completely independent and don't interfere with each other's variable locations. Thus storage and transient storage state variables can be safely interleaved without any side effects. Only value types are supported for transient storage.

State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings (see below), data is stored contiguously item after item starting with the first state variable, which is stored in slot `0`. For each variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Value types use only as many bytes as are necessary to store them.
- If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot.
- Structs and array data always start a new slot and their items are packed tightly according to these rules.
- Items following struct or array data always start a new storage slot.

For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

The elements of structs and arrays are stored after each other, just as if they were given as individual values.

If a contract specifies a *custom storage layout*, the slots assigned to static storage variables are shifted according the value defined as the layout base. Locations of dynamic arrays and mappings are also indirectly affected by this due to

shifting of the static slots they are based on. The custom layout is specified in the most derived contract and, following the order explained above, starting from the most base-ward contract's variables, all storage slots are adjusted.

In the following example, contract `C` inherits from contracts `A` and `B` and also specifies a custom storage base slot. The result is that all storage variable slots of the inheritance tree are adjusted according to the value specified by `C`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.29;

struct S {
    int32 x;
    bool y;
}

contract A {
    uint a;
    uint128 transient b;
    uint constant c = 10;
    uint immutable d = 12;
}

contract B {
    uint8[] e;
    mapping(uint => S) f;
    uint16 g;
    uint16 h;
    bytes16 transient i;
    S s;
    int8 k;
}

contract C is A, B layout at 42 {
    bytes21 l;
    uint8[10] m;
    bytes5[8] n;
    bytes5 o;
}
```

In the example, the storage layout starts with the inherited state variable `a` stored directly inside the base slot (slot `42`). Transient, constant and immutable variables are stored in separate locations, and thus, `b`, `i`, `c` and `d` have no effect on the storage layout. Then we get to the dynamic array `e` and mapping `f`. They both reserve a whole slot whose address will be used to *calculate* the location where their data is actually stored. The slot cannot be shared with any other variable, because the resulting addresses must be unique. The next two variables, `g` and `h`, need 2 bytes each and can be packed together into slot `45`, at offsets `0` and `2` respectively. Since `s` is a struct, its two members are packed contiguously, each taking up 5 bytes. Even though they both would still fit in slot `45`, structs and arrays always start a new slot. Therefore, `s` is placed in slot `46` and the next variable, `k`, in slot `47`. Base contracts, on the other hand, can share slots with derived ones, so `l` does not require an new one. Then variable `m`, which is an array of 10 items, gets into slot `48` and takes up 10 bytes. `n` is an array as well, but due to the size of its items, cannot fill its first slot perfectly and spills over to the next one. Finally, variable `o` ends up in slot `51`, even though it is of the same type as items of `n`. As explained before, variables after structs and arrays always start a new slot.

Putting it all together, the storage and transient storage layouts of contract `C` can be illustrated as follows:

- Storage:

```
42 [aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa]
43 [eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee]
44 [ffffffffffffffffffffffffffffffff]
45 [                              hhgg]
46 [                             yxxxx]
47 [          llllllllllllllllllllk]
48 [                         mmmmmmmmmm]
49 [  nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn]
50 [                         nnnnnnnnnn]
51 [                              ooooo]
```

- Transient storage:

```
00 [iiiiiiiiiiiiiiiibbbbbbbbbbbbbbbb]
```

Note that the storage specifier affects `A` and `B` only as a part of `C`'s inheritance hierarchy. When deployed independently, their storage starts at `0`:

- Storage layout of `A`:

```
00 [aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa]
```

- Storage layout of `B`:

```
00 [eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee]
01 [ffffffffffffffffffffffffffffffff]
02 [                              hhgg]
03 [                             yxxxx]
04 [                                 k]
```

Warning

When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

It might be beneficial to use reduced-size types if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. If you are not reading or writing all the values in a slot at the same time, this can have the opposite effect, though: When one value is written to a multi-value storage slot, the storage slot has to be read first and then combined with the new value such that other data in the same slot is not destroyed.

When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.

Finally, in order to allow the EVM to optimize for this, ensure that you try to order your storage variables and `struct` members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

Note

The layout of state variables in storage is considered to be part of the external interface of Solidity due to the fact

that storage pointers can be passed to libraries. This means that any change to the rules outlined in this section is considered a breaking change of the language and due to its critical nature should be considered very carefully before being executed. In the event of such a breaking change, we would want to release a compatibility mode in which the compiler would generate bytecode supporting the old layout.

3.16.1 Mappings and Dynamic Arrays

Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored "in between" the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the *rules above* and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

Assume the storage location of the mapping or array ends up being a slot `p` after applying *the storage layout rules*. For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see *below*). For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations.

Array data is located starting at `keccak256(p)` and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively. The location of element `x[i][j]`, where the type of `x` is `uint24[][]`, is computed as follows (again, assuming `x` itself is stored at slot `p`): The slot is `keccak256(keccak256(p) + i) + floor(j / floor(256 / 24))` and the element can be obtained from the slot data `v` using `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`.

The value corresponding to a mapping key `k` is located at `keccak256(h(k) . p)` where `.` is concatenation and `h` is a function that is applied to the key depending on its type:

- for value types, `h` pads the value to 32 bytes in the same way as when storing the value in memory.
- for strings and byte arrays, `h(k)` is just the unpadded data.

If the mapping value is a non-value type, the computed slot marks the start of the data. If the value is of struct type, for example, you have to add an offset corresponding to the struct member to reach the member.

As an example, consider the following contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;


contract C {
    struct S { uint16 a; uint16 b; uint256 c; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

Let us compute the storage location of `data[4][9].c`. The position of the mapping itself is `1` (the variable `x` with 32 bytes precedes it). This means `data[4]` is stored at `keccak256(uint256(4) . uint256(1))`. The type of `data[4]` is again a mapping and the data for `data[4][9]` starts at slot `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))`. The slot offset of the member `c` inside the struct `S` is `1` because `a` and `b` are packed in a single slot. This means the slot for `data[4][9].c` is `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`. The type of the value is `uint256`, so it uses a single slot.

`bytes` and `string`

`bytes` and `string` are encoded identically. In general, the encoding is similar to `bytes1[]`, in the sense that there is a slot for the array itself and a data area that is computed using a `keccak256` hash of that slot's position. However, for short values (shorter than 32 bytes) the array elements are stored together with the length in the same slot.

In particular: if the data is at most `31` bytes long, the elements are stored in the higher-order bytes (left aligned) and the lowest-order byte stores the value `length * 2`. For byte arrays that store data which is `32` or more bytes long, the main slot `p` stores `length * 2 + 1` and the data is stored as usual in `keccak256(p)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set: short (not set) and long (set).

Note

Handling invalidly encoded slots is currently not supported but may be added in the future. If you are compiling via IR, reading an invalidly encoded slot results in a `Panic(0x22)` error.

3.16.2 JSON Output

The storage (or transient storage) layout of a contract can be requested via the *standard JSON interface*. The output is a JSON object containing two keys, `storage` and `types`. The `storage` object is an array where each element has the following form:

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

The example above is the storage layout of `contract A { uint x; }` from source unit `fileA` and

- `astId` is the id of the AST node of the state variable's declaration
- `contract` is the name of the contract including its path as prefix
- `label` is the name of the state variable
- `offset` is the offset in bytes within the storage slot according to the encoding
- `slot` is the storage slot where the state variable resides or starts. This number may be very large and therefore its JSON value is represented as a string.
- `type` is an identifier used as key to the variable's type information (described in the following)

The given `type`, in this case `t_uint256` represents an element in `types`, which has the form:

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

where

- `encoding` how the data is encoded in storage, where the possible values are:
  - `inplace`: data is laid out contiguously in storage (see *above*).

  - `mapping`: Keccak-256 hash-based method (see *above*).
  - `dynamic_array`: Keccak-256 hash-based method (see *above*).
  - `bytes`: single slot or Keccak-256 hash-based depending on the data size (see *above*).
- `label` is the canonical type name.
- `numberOfBytes` is the number of used bytes (as a decimal string). Note that if `numberOfBytes > 32` this means that more than one slot is used.

Some types have extra information besides the four above. Mappings contain its `key` and `value` types (again referencing an entry in this mapping of types), arrays have its `base` type, and structs list their `members` in the same format as the top-level `storage` (see *above*).

Note

The JSON output format of a contract's storage layout is still considered experimental and is subject to change in non-breaking releases of Solidity.

The following example shows a contract and both its storage and transient storage layout, containing value and reference types, types that are encoded packed, and nested types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.28;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint transient y;
    uint w;
    uint transient z;

    S s;
    address addr;
    address transient taddr;
    mapping(uint => mapping(address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```

Storage Layout

```
{
  "storage": [
    {
      "astId": 15,
      "contract": "fileA:A",
```

(continues on next page)

(continued from previous page)


```
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
      "astId": 19,
      "contract": "fileA:A",
      "label": "w",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 24,
      "contract": "fileA:A",
      "label": "s",
      "offset": 0,
      "slot": "2",
      "type": "t_struct(S)13_storage"
    },
    {
      "astId": 26,
      "contract": "fileA:A",
      "label": "addr",
      "offset": 0,
      "slot": "6",
      "type": "t_address"
    },
    {
      "astId": 34,
      "contract": "fileA:A",
      "label": "map",
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
    {
      "astId": 37,
      "contract": "fileA:A",
      "label": "array",
      "offset": 0,
      "slot": "8",
      "type": "t_array(t_uint256)dyn_storage"
    },
    {
      "astId": 39,
      "contract": "fileA:A",
      "label": "s1",
      "offset": 0,
      "slot": "9",
      "type": "t_string_storage"
```


(continues on next page)

(continued from previous page)

```
    },
    {
      "astId": 41,
      "contract": "fileA:A",
      "label": "b1",
      "offset": 0,
      "slot": "10",
      "type": "t_bytes_storage"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_array(t_uint256)2_storage": {
      "base": "t_uint256",
      "encoding": "inplace",
      "label": "uint256[2]",
      "numberOfBytes": "64"
    },
    "t_array(t_uint256)dyn_storage": {
      "base": "t_uint256",
      "encoding": "dynamic_array",
      "label": "uint256[]",
      "numberOfBytes": "32"
    },
    "t_bool": {
      "encoding": "inplace",
      "label": "bool",
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
    },
    "t_mapping(t_address,t_bool)": {
      "encoding": "mapping",
      "key": "t_address",
      "label": "mapping(address => bool)",
      "numberOfBytes": "32",
      "value": "t_bool"
    },
    "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
      "encoding": "mapping",
      "key": "t_uint256",
      "label": "mapping(uint256 => mapping(address => bool))",
      "numberOfBytes": "32",
      "value": "t_mapping(t_address,t_bool)"
    },
```

(continues on next page)

(continued from previous page)


```
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_struct(S)13_storage": {
      "encoding": "inplace",
      "label": "struct A.S",
      "members": [
        {
          "astId": 3,
          "contract": "fileA:A",
          "label": "a",
          "offset": 0,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 5,
          "contract": "fileA:A",
          "label": "b",
          "offset": 16,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 9,
          "contract": "fileA:A",
          "label": "staticArray",
          "offset": 0,
          "slot": "1",
          "type": "t_array(t_uint256)2_storage"
        },
        {
          "astId": 12,
          "contract": "fileA:A",
          "label": "dynArray",
          "offset": 0,
          "slot": "3",
          "type": "t_array(t_uint256)dyn_storage"
        }
      ],
      "numberOfBytes": "128"
    },
    "t_uint128": {
      "encoding": "inplace",
      "label": "uint128",
      "numberOfBytes": "16"
    },
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
```


(continues on next page)

(continued from previous page)

```
      "numberOfBytes": "32"
    }
  }
}
```

Transient Storage Layout

```
{
  "storage": [
    {
      "astId": 17,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
      "astId": 21,
      "contract": "fileA:A",
      "label": "z",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 28,
      "contract": "fileA:A",
      "label": "taddr",
      "offset": 0,
      "slot": "2",
      "type": "t_address"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
      "numberOfBytes": "32"
    }
  }
}
```

3.17 Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

- `0x00` - `0x3f` (64 bytes): scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes): currently allocated memory size (aka. free memory pointer)
- `0x60` - `0x7f` (32 bytes): zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to `0x80` initially).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `bytes1[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

Warning

There are some operations in Solidity that need a temporary memory area larger than 64 bytes and therefore will not fit into the scratch space. They will be placed where the free memory points to, but given their short lifetime, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one should not expect the free memory to point to zeroed out memory.

While it may seem like a good idea to use `msize` to arrive at a definitely zeroed out memory area, using such a pointer non-temporarily without updating the free memory pointer can have unexpected results.

3.17.1 Differences to Layout in Storage

As described above the layout in memory is different from the layout in *storage*. Below there are some examples.

Example for Difference in Arrays

The following array occupies 32 bytes (1 slot) in storage, but 128 bytes (4 items with 32 bytes each) in memory.

```
uint8[4] a;
```

Example for Difference in Struct Layout

The following struct occupies 96 bytes (3 slots of 32 bytes) in storage, but 128 bytes (4 items with 32 bytes each) in memory.

```
struct S {
    uint a;
    uint b;
    uint8 c;
    uint8 d;
}
```

3.18 Layout of Call Data

The input data for a function call is assumed to be in the format defined by the *ABI specification*. Among others, the ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Arguments for the constructor of a contract are directly appended at the end of the contract's code, also in ABI encoding. The constructor will access them through a hard-coded offset, and not by using the `codesize` opcode, since this of course changes when appending data to the code.

3.19 Cleaning Up Variables

Ultimately, all values in the EVM are stored in 256 bit words. Thus, in some cases, when the type of a value has less than 256 bits, it is necessary to clean the remaining bits. The Solidity compiler is designed to do such cleaning before any operations that might be adversely affected by the potential garbage in the remaining bits. For example, before writing a value to memory, the remaining bits need to be cleared because the memory contents can be used for computing hashes or sent as the data of a message call. Similarly, before storing a value in the storage, the remaining bits need to be cleaned because otherwise the garbled value can be observed.

Note that access via inline assembly is not considered such an operation: If you use inline assembly to access Solidity variables shorter than 256 bits, the compiler does not guarantee that the value is properly cleaned up.

Moreover, we do not clean the bits if the immediately following operation is not affected. For instance, since any non-zero value is considered `true` by `JUMPI` instruction, we do not clean the boolean values before they are used as the condition for `JUMPI`.

In addition to the design principle above, the Solidity compiler cleans input data when it is loaded onto the stack.

The following table describes the cleaning rules applied to different types, where `higher bits` refers to the remaining bits in case the type has less than 256 bits.

Type	Valid Values	Cleanup of Invalid Values
enum of n members	0 until n - 1	throws exception
bool	0 or 1	results in 1
signed integers	higher bits set to the sign bit	currently silently signextends to a valid value, i.e. all higher bits are set to the sign bit; may throw an exception in the future
unsigned integers	higher bits zeroed	currently silently masks to a valid value, i.e. all higher bits are set to zero; may throw an exception in the future

Note that valid and invalid values are dependent on their type size. Consider `uint8`, the unsigned 8-bit type, which has the following valid values:

```
0000...0000 0000 0000
0000...0000 0000 0001
0000...0000 0000 0010
....
0000...0000 1111 1111
```

Any invalid value will have the higher bits set to zero:

```
0101...1101 0010 1010   invalid value
0000...0000 0010 1010   cleaned value
```

For `int8`, the signed 8-bit type, the valid values are:

Negative

```
1111...1111 1111 1111
1111...1111 1111 1110
....
1111...1111 1000 0000
```

Positive

```
0000...0000 0000 0000
0000...0000 0000 0001
0000...0000 0000 0010
....
0000...0000 0111 1111
```

The compiler will `signextend` the sign bit, which is 1 for negative and 0 for positive values, overwriting the higher bits:

Negative

```
0010...1010 1111 1111   invalid value
1111...1111 1111 1111   cleaned value
```

Positive

```
1101...0101 0000 0100   invalid value
0000...0000 0000 0100   cleaned value
```

3.20 Source Mappings

As part of the AST output, the compiler provides the range of the source code that is represented by the respective node in the AST. This can be used for various purposes ranging from static analysis tools that report errors based on the AST and debugging tools that highlight local variables and their uses.

Furthermore, the compiler can also generate a mapping from the bytecode to the range in the source code that generated the instruction. This is again important for static analysis tools that operate on bytecode level and for displaying the current position in the source code inside a debugger or for breakpoint handling. This mapping also contains other information, like the jump type and the modifier depth (see below).

Both kinds of source mappings use integer identifiers to refer to source files. The identifier of a source file is stored in `output['sources'][sourceName]['id']` where `output` is the output of the standard-json compiler interface parsed as JSON. For some utility routines, the compiler generates "internal" source files that are not part of the original input but are referenced from the source mappings. These source files together with their identifiers can be obtained via `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']`.

Note

In the case of instructions that are not associated with any particular source file, the source mapping assigns an integer identifier of `-1`. This may happen for bytecode sections stemming from compiler-generated inline assembly statements.

The source mappings inside the AST use the following notation:

`s:l:f`

Where `s` is the byte-offset to the start of the range in the source file, `l` is the length of the source range in bytes and `f` is the source index mentioned above.

The encoding in the source mapping for the bytecode is more complicated: It is a list of `s:l:f:j:m` separated by `;`. Each of these elements corresponds to an instruction, i.e. you cannot use the byte offset but have to use the instruction offset (push instructions are longer than a single byte). The fields `s`, `l` and `f` are as above. `j` can be either `i`, `o` or `-` signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of e.g. a loop. The last field, `m`, is an integer that denotes the "modifier depth". This depth is increased whenever the placeholder statement (_) is entered in a modifier and decreased when it is left again. This allows debuggers to track tricky cases like the same modifier being used twice or multiple placeholder statements being used in a single modifier.

In order to compress these source mappings especially for bytecode, the following rules are used:

- If a field is empty, the value of the preceding element is used.
- If a `:` is missing, all following fields are considered empty.

This means the following source mappings represent the same information:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2:1:2;;`

Important to note is that when the *verbatim* builtin is used, the source mappings will be invalid: The builtin is considered a single instruction instead of potentially multiple.

3.21 The Optimizer

The Solidity compiler involves optimizations at three different levels (in order of execution):

- Optimizations during code generation based on a direct analysis of Solidity code.
- Optimizing transformations on the Yul IR code.
- Optimizations at the opcode level.

The opcode-based optimizer applies a set of simplification rules to opcodes. It also combines equal code sets and removes unused code.

The Yul-based optimizer is much more powerful, because it can work across function calls. For example, arbitrary jumps are not possible in Yul, so it is possible to compute the side-effects of each function. Consider two function calls, where the first does not modify storage and the second does modify storage. If their arguments and return values do not depend on each other, we can reorder the function calls. Similarly, if a function is side-effect free and its result is multiplied by zero, you can remove the function call completely.

The codegen-based optimizer affects the initial low-level code produced from the Solidity input. In the legacy pipeline, the bytecode is generated immediately and most of the optimizations of this kind are implicit and not configurable, the only exception being an optimization which changes the order of literals in binary operations. The IR-based pipeline takes a different approach and produces Yul IR closely matching the structure of the Solidity code, with nearly all optimizations deferred to the Yul optimizer module. In that case codegen-level optimization is done only in very limited cases which are difficult to handle in Yul IR, but are straightforward with the high-level information from analysis phase at hand. An example of such an optimization is the bypass of checked arithmetic when incrementing the counter in certain idiomatic `for` loops.

Currently, the parameter `--optimize` activates the opcode-based optimizer for the generated bytecode and the Yul optimizer for the Yul code generated internally, for example for ABI coder v2. One can use `solc --ir-optimized --optimize` to produce an optimized Yul IR for a Solidity source. Similarly, one can use `solc --strict-assembly --optimize` for a stand-alone Yul mode.

Note

Some optimizer steps, such as, for example, the peephole optimizer and the *unchecked loop increment optimizer* are always enabled by default and can only be turned off via the *Standard JSON*.

Note

An empty optimizer sequence, i.e `:`, is accepted even without `--optimize` in order to fully disable the user-supplied portion of the Yul *optimizer sequence*, as by default, even when the optimizer is not turned on, the *unused pruner* step will be run.

You can find more details on both optimizer modules and their optimization steps below.

3.21.1 Benefits of Optimizing Solidity Code

Overall, the optimizer tries to simplify complicated expressions, which reduces both code size and execution cost, i.e., it can reduce gas needed for contract deployment as well as for external calls made to the contract. It also specializes or inlines functions. Especially function inlining is an operation that can cause much bigger code, but it is often done because it results in opportunities for more simplifications.

3.21.2 Differences between Optimized and Non-Optimized Code

Generally, the most visible difference is that constant expressions are evaluated at compile time. When it comes to the ASM output, one can also notice a reduction of equivalent or duplicate code blocks (compare the output of the flags `--asm` and `--asm --optimize`). However, when it comes to the Yul/intermediate-representation, there can be significant differences, for example, functions may be inlined, combined, or rewritten to eliminate redundancies, etc. (compare the output between the flags `--ir` and `--optimize --ir-optimized`).

3.21.3 Optimizer Parameter Runs

The number of runs (`--optimize-runs`) specifies roughly how often each opcode of the deployed code will be executed across the life-time of the contract. This means it is a trade-off parameter between code size (deploy cost) and code execution cost (cost after deployment). A "runs" parameter of "1" will produce short but expensive code. In contrast, a larger "runs" parameter will produce longer but more gas efficient code. The maximum value of the parameter is `2**32-1`.

Note

A common misconception is that this parameter specifies the number of iterations of the optimizer. This is not true: The optimizer will always run as many times as it can still improve the code.

3.21.4 Opcode-Based Optimizer Module

The opcode-based optimizer module operates on assembly code. It splits the sequence of instructions into basic blocks at `JUMPs` and `JUMPDESTs`. Inside these blocks, the optimizer analyzes the instructions and records every modification to the stack, memory, or storage as an expression which consists of an instruction and a list of arguments which are pointers to other expressions.

Additionally, the opcode-based optimizer uses a component called "CommonSubexpressionEliminator" that, amongst other tasks, finds expressions that are always equal (on every input) and combines them into an expression class. It first tries to find each new expression in a list of already known expressions. If no such matches are found, it simplifies

the expression according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is a recursive process, we can also apply the latter rule if the second factor is a more complex expression which we know always evaluates to one.

Certain optimizer steps symbolically track the storage and memory locations. For example, this information is used to compute Keccak-256 hashes that can be evaluated during compile time. Consider the sequence:

```
PUSH 32
PUSH 0
CALLDATALOAD
PUSH 100
DUP2
MSTORE
KECCAK256
```

or the equivalent Yul

```
let x := calldataload(0)
mstore(x, 100)
let value := keccak256(x, 32)
```

In this case, the optimizer tracks the value at a memory location `calldataload(0)` and then realizes that the Keccak-256 hash can be evaluated at compile time. This only works if there is no other instruction that modifies memory between the `mstore` and `keccak256`. So if there is an instruction that writes to memory (or storage), then we need to erase the knowledge of the current memory (or storage). There is, however, an exception to this erasing, when we can easily see that the instruction doesn't write to a certain location.

For example,

```
let x := calldataload(0)
mstore(x, 100)
// Current knowledge memory location x -> 100
let y := add(x, 32)
// Does not clear the knowledge that x -> 100, since y does not write to [x, x + 32)
mstore(y, 200)
// This Keccak-256 can now be evaluated
let value := keccak256(x, 32)
```

Therefore, modifications to storage and memory locations, of say location `l`, must erase knowledge about storage or memory locations which may be equal to `l`. More specifically, for storage, the optimizer has to erase all knowledge of symbolic locations, that may be equal to `l` and for memory, the optimizer has to erase all knowledge of symbolic locations that may not be at least 32 bytes away. If `m` denotes an arbitrary location, then this decision on erasure is done by computing the value `sub(l, m)`. For storage, if this value evaluates to a literal that is non-zero, then the knowledge about `m` will be kept. For memory, if the value evaluates to a literal that is between `32` and `2**256 - 32`, then the knowledge about `m` will be kept. In all other cases, the knowledge about `m` will be erased.

After this process, we know which expressions have to be on the stack at the end, and have a list of modifications to memory and storage. This information is stored together with the basic blocks and is used to link them. Furthermore, knowledge about the stack, storage and memory configuration is forwarded to the next block(s).

If we know the targets of all `JUMP` and `JUMPI` instructions, we can build a complete control flow graph of the program. If there is only one target we do not know (this can happen as in principle, jump targets can be computed from inputs), we have to erase all knowledge about the input state of a block as it can be the target of the unknown `JUMP`. If the opcode-based optimizer module finds a `JUMPI` whose condition evaluates to a constant, it transforms it to an unconditional jump.

As the last step, the code in each block is re-generated. The optimizer creates a dependency graph from the expressions

on the stack at the end of the block, and it drops every operation that is not part of this graph. It generates code that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed). Finally, it generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a `JUMPI` and during the analysis, the condition evaluates to a constant, the `JUMPI` is replaced based on the value of the constant. Thus code like

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2) // this condition is never true
  return 2;
else
  return 1;
```

simplifies to this:

```
data[7] = 9;
return 1;
```

Simple Inlining

Since Solidity version 0.8.2, there is another optimizer step that replaces certain jumps to blocks containing "simple" instructions ending with a "jump" by a copy of these instructions. This corresponds to inlining of simple, small Solidity or Yul functions. In particular, the sequence `PUSHTAG(tag) JUMP` may be replaced, whenever the `JUMP` is marked as jump "into" a function and behind `tag` there is a basic block (as described above for the "CommonSubexpressionEliminator") that ends in another `JUMP` which is marked as a jump "out of" a function.

In particular, consider the following prototypical example of assembly generated for a call to an internal Solidity function:

```
  tag_return
  tag_f
  jump      // in
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

As long as the body of the function is a continuous basic block, the "Inliner" can replace `tag_f jump` by the block at `tag_f` resulting in:

```
  tag_return
  ...body of function f...
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

Now ideally, the other optimizer steps described above will result in the return tag push being moved towards the remaining jump resulting in:

```
  ...body of function f...
  tag_return
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

In this situation the "PeepholeOptimizer" will remove the return jump. Ideally, all of this can be done for all references to `tag_f` leaving it unused, s.t. it can be removed, yielding:

```
...body of function f...
...opcodes after call to f...
```

So the call to function `f` is inlined and the original definition of `f` can be removed.

Inlining like this is attempted, whenever a heuristics suggests that inlining is cheaper over the lifetime of a contract than not inlining. This heuristics depends on the size of the function body, the number of other references to its tag (approximating the number of calls to the function) and the expected number of executions of the contract (the global optimizer parameter "runs").

3.21.5 Yul-Based Optimizer Module

The Yul-based optimizer consists of several stages and components that all transform the AST in a semantically equivalent way. The goal is to end up either with code that is shorter or at least only marginally longer but will allow further optimization steps.

> **Warning**
>
> Since the optimizer is under heavy development, the information here might be outdated. If you rely on a certain functionality, please reach out to the team directly.

The optimizer currently follows a purely greedy strategy and does not do any backtracking.

All components of the Yul-based optimizer module are explained below. The following transformation steps are the main components:

- SSATransform
- CommonSubexpressionEliminator
- ExpressionSimplifier
- UnusedAssignEliminator
- FullInliner

Optimizer Steps

This is a list of all steps the Yul-based optimizer sorted alphabetically. You can find more information on the individual steps and their sequence below.

Abbreviation	Full name
`f`	*BlockFlattener*
`l`	*CircularReferencesPruner*
`c`	*CommonSubexpressionEliminator*
`C`	*ConditionalSimplifier*
`U`	*ConditionalUnsimplifier*
`n`	*ControlFlowSimplifier*
`D`	*DeadCodeEliminator*
`E`	*EqualStoreEliminator*
`v`	*EquivalentFunctionCombiner*
`e`	*ExpressionInliner*
`j`	*ExpressionJoiner*
`s`	*ExpressionSimplifier*
`x`	*ExpressionSplitter*
`I`	*ForLoopConditionIntoBody*
`O`	*ForLoopConditionOutOfBody*
`o`	*ForLoopInitRewriter*
`i`	*FullInliner*
`g`	*FunctionGrouper*
`h`	*FunctionHoister*
`F`	*FunctionSpecializer*
`T`	*LiteralRematerialiser*
`L`	*LoadResolver*
`M`	*LoopInvariantCodeMotion*
`m`	*Rematerialiser*
`V`	*SSAReverser*
`a`	*SSATransform*
`t`	*StructuralSimplifier*
`r`	*UnusedAssignEliminator*
`p`	*UnusedFunctionParameterPruner*
`S`	*UnusedStoreEliminator*
`u`	*UnusedPruner*
`d`	*VarDeclInitializer*

Some steps depend on properties ensured by `BlockFlattener`, `FunctionGrouper`, `ForLoopInitRewriter`. For this reason the Yul optimizer always applies them before applying any steps supplied by the user.

Selecting Optimizations

By default the optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override this sequence and supply your own using the `--yul-optimizations` option:

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul:fDnTOcmu'
```

The order of steps is significant and affects the quality of the output. Moreover, applying a step may uncover new optimization opportunities for others that were already applied, so repeating steps is often beneficial.

The sequence inside `[...]` will be applied multiple times in a loop until the Yul code remains unchanged or until the maximum number of rounds (currently 12) has been reached. Brackets (`[]`) may be used multiple times in a sequence,

but can not be nested.

An important thing to note, is that there are some hardcoded steps that are always run before and after the user-supplied sequence, or the default sequence if one was not supplied by the user.

The cleanup sequence delimiter `:` is optional, and is used to supply a custom cleanup sequence in order to replace the default one. If omitted, the optimizer will simply apply the default cleanup sequence. In addition, the delimiter may be placed at the beginning of the user-supplied sequence, which will result in the optimization sequence being empty, whereas conversely, if placed at the end of the sequence, will be treated as an empty cleanup sequence.

Preprocessing

The preprocessing components perform transformations to get the program into a certain normal form that is easier to work with. This normal form is kept during the rest of the optimization process.

Disambiguator

The disambiguator takes an AST and returns a fresh copy where all identifiers have unique names in the input AST. This is a prerequisite for all other optimizer stages. One of the benefits is that identifier lookup does not need to take scopes into account which simplifies the analysis needed for other steps.

All subsequent stages have the property that all names stay unique. This means if a new identifier needs to be introduced, a new unique name is generated.

FunctionHoister

The function hoister moves all function definitions to the end of the topmost block. This is a semantically equivalent transformation as long as it is performed after the disambiguation stage. The reason is that moving a definition to a higher-level block cannot decrease its visibility and it is impossible to reference variables defined in a different function.

The benefit of this stage is that function definitions can be looked up more easily and functions can be optimized in isolation without having to traverse the AST completely.

FunctionGrouper

The function grouper has to be applied after the Disambiguator and the FunctionHoister. Its effect is that all topmost elements that are not function definitions are moved into a single block which is the first statement of the root block.

After this step, a program has the following normal form:

```
{ I F... }
```

Where `I` is a (potentially empty) block that does not contain any function definitions (not even recursively) and `F` is a list of function definitions such that no function contains a function definition.

The benefit of this stage is that we always know where the list of functions begins.

ForLoopConditionIntoBody

This transformation moves the loop-iteration condition of a `for` loop into loop body. We need this transformation because *ExpressionSplitter* will not apply to iteration condition expressions (the `C` in the following example).

```
for { Init... } C { Post... } {
    Body...
}
```

is transformed to

```
for { Init... } 1 { Post... } {
    if iszero(C) { break }
    Body...
}
```

This transformation can also be useful when paired with LoopInvariantCodeMotion, since invariants in the loop-invariant conditions can then be taken outside the loop.

ForLoopInitRewriter

This transformation moves the initialization part of a `for` loop to before the loop:

```
for { Init... } C { Post... } {
    Body...
}
```

is transformed to

```
Init...
for {} C { Post... } {
    Body...
}
```

This eases the rest of the optimization process because we can ignore the complicated scoping rules of the `for` loop initialization block.

VarDeclInitializer

This step rewrites variable declarations so that all of them are initialized. Declarations like `let x, y` are split into multiple declaration statements.

Only supports initializing with the zero literal for now.

Pseudo-SSA Transformation

The purpose of this components is to get the program into a longer form, so that other components can more easily work with it. The final representation will be similar to a static-single-assignment (SSA) form, with the difference that it does not make use of explicit "phi" functions which combines the values from different branches of control flow because such a feature does not exist in the Yul language. Instead, when control flow merges, if a variable is re-assigned in one of the branches, a new SSA variable is declared to hold its current value, so that the following expressions still only need to reference SSA variables.

An example transformation is the following:

```
{
    let a := calldataload(0)
    let b := calldataload(0x20)
    if gt(a, 0) {
        b := mul(b, 0x20)
    }
    a := add(a, 1)
    sstore(a, add(b, 0x20))
}
```

When all the following transformation steps are applied, the program will look as follows:

```
{
    let _1 := 0
    let a_9 := calldataload(_1)
    let a := a_9
    let _2 := 0x20
    let b_10 := calldataload(_2)
    let b := b_10
    let _3 := 0
    let _4 := gt(a_9, _3)
    if _4
    {
        let _5 := 0x20
        let b_11 := mul(b_10, _5)
        b := b_11
    }
    let b_12 := b
    let _6 := 1
    let a_13 := add(a_9, _6)
    let _7 := 0x20
    let _8 := add(b_12, _7)
    sstore(a_13, _8)
}
```

Note that the only variable that is re-assigned in this snippet is `b`. This re-assignment cannot be avoided because `b` has different values depending on the control flow. All other variables never change their value once they are defined. The advantage of this property is that variables can be freely moved around and references to them can be exchanged by their initial value (and vice-versa), as long as these values are still valid in the new context.

Of course, the code here is far from being optimized. To the contrary, it is much longer. The hope is that this code will be easier to work with and furthermore, there are optimizer steps that undo these changes and make the code more compact again at the end.

ExpressionSplitter

The expression splitter turns expressions like `add(mload(0x123), mul(mload(0x456), 0x20))` into a sequence of declarations of unique variables that are assigned sub-expressions of that expression so that each function call has only variables as arguments.

The above would be transformed into

```
{
    let _1 := 0x20
    let _2 := 0x456
    let _3 := mload(_2)
    let _4 := mul(_3, _1)
    let _5 := 0x123
    let _6 := mload(_5)
    let z := add(_6, _4)
}
```

Note that this transformation does not change the order of opcodes or function calls.

It is not applied to loop iteration-condition, because the loop control flow does not allow this "outlining" of the inner expressions in all cases. We can sidestep this limitation by applying *ForLoopConditionIntoBody* to move the iteration condition into loop body.

The final program should be in an *expression-split form*, where (with the exception of loop conditions) function calls cannot appear nested inside expressions and all function call arguments have to be variables.

The benefits of this form are that it is much easier to re-order the sequence of opcodes and it is also easier to perform function call inlining. Furthermore, it is simpler to replace individual parts of expressions or re-organize the "expression tree". The drawback is that such code is much harder to read for humans.

SSATransform

This stage tries to replace repeated assignments to existing variables by declarations of new variables as much as possible. The reassignments are still there, but all references to the reassigned variables are replaced by the newly declared variables.

Example:

```
{
    let a := 1
    mstore(a, 2)
    a := 3
}
```

is transformed to

```
{
    let a_1 := 1
    let a := a_1
    mstore(a_1, 2)
    let a_3 := 3
    a := a_3
}
```

Exact semantics:

For any variable `a` that is assigned to somewhere in the code (variables that are declared with value and never re-assigned are not modified) perform the following transforms:

- replace `let a := v` by `let a_i := v let a := a_i`
- replace `a := v` by `let a_i := v a := a_i` where `i` is a number such that `a_i` is yet unused.

Furthermore, always record the current value of `i` used for `a` and replace each reference to `a` by `a_i`. The current value mapping is cleared for a variable `a` at the end of each block in which it was assigned to and at the end of the `for` loop init block if it is assigned inside the `for` loop body or post block. If a variable's value is cleared according to the rule above and the variable is declared outside the block, a new SSA variable will be created at the location where control flow joins, this includes the beginning of loop post/body block and the location right after `if`/`switch`/`for`/block statement.

After this stage, the UnusedAssignEliminator is recommended to remove the unnecessary intermediate assignments.

This stage provides best results if the ExpressionSplitter and the CommonSubexpressionEliminator are run right before it, because then it does not generate excessive amounts of variables. On the other hand, the CommonSubexpressionEliminator could be more efficient if run after the SSA transform.

UnusedAssignEliminator

The SSA transform always generates an assignment of the form `a := a_i`, even though these might be unnecessary in many cases, like the following example:

```
{
    let a := 1
    a := mload(a)
    a := sload(a)
    sstore(a, 1)
}
```

The SSA transform converts this snippet to the following:

```
{
    let a_1 := 1
    let a := a_1
    let a_2 := mload(a_1)
    a := a_2
    let a_3 := sload(a_2)
    a := a_3
    sstore(a_3, 1)
}
```

The UnusedAssignEliminator removes all the three assignments to `a`, because the value of `a` is not used and thus turn this snippet into strict SSA form:

```
{
    let a_1 := 1
    let a_2 := mload(a_1)
    let a_3 := sload(a_2)
    sstore(a_3, 1)
}
```

Of course the intricate parts of determining whether an assignment is unused or not are connected to joining control flow.

The component works as follows in detail:

The AST is traversed twice: in an information gathering step and in the actual removal step. During information gathering, we maintain a mapping from assignment statements to the three states "unused", "undecided" and "used" which signifies whether the assigned value will be used later by a reference to the variable.

When an assignment is visited, it is added to the mapping in the "undecided" state (see remark about `for` loops below) and every other assignment to the same variable that is still in the "undecided" state is changed to "unused". When a variable is referenced, the state of any assignment to that variable still in the "undecided" state is changed to "used".

At points where control flow splits, a copy of the mapping is handed over to each branch. At points where control flow joins, the two mappings coming from the two branches are combined in the following way: Statements that are only in one mapping or have the same state are used unchanged. Conflicting values are resolved in the following way:

- "unused", "undecided" -> "undecided"
- "unused", "used" -> "used"
- "undecided", "used" -> "used"

For `for` loops, the condition, body and post-part are visited twice, taking the joining control-flow at the condition into account. In other words, we create three control flow paths: Zero runs of the loop, one run and two runs and then combine them at the end.

Simulating a third run or even more is unnecessary, which can be seen as follows:

A state of an assignment at the beginning of the iteration will deterministically result in a state of that assignment at the end of the iteration. Let this state mapping function be called `f`. The combination of the three different states `unused`, `undecided` and `used` as explained above is the `max` operation where `unused = 0`, `undecided = 1` and `used = 2`.

The proper way would be to compute

```
max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

as state after the loop. Since `f` just has a range of three different values, iterating it has to reach a cycle after at most three iterations, and thus `f(f(f(s)))` has to equal one of `s`, `f(s)`, or `f(f(s))` and thus

```
max(s, f(s), f(f(s))) = max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

In summary, running the loop at most twice is enough because there are only three different states.

For `switch` statements that have a default case, there is no control-flow part that skips the `switch`.

When a variable goes out of scope, all statements still in the "undecided" state are changed to "unused", unless the variable is the return parameter of a function - there, the state changes to "used".

In the second traversal, all assignments that are in the "unused" state are removed.

This step is usually run right after the SSA transform to complete the generation of the pseudo-SSA.

Tools

Movability

Movability is a property of an expression. It roughly means that the expression is side-effect free and its evaluation only depends on the values of variables and the call-constant state of the environment. Most expressions are movable. The following parts make an expression non-movable:

- function calls (might be relaxed in the future if all statements in the function are movable)
- opcodes that (can) have side-effects (like `call` or `selfdestruct`)
- opcodes that read or write memory, storage or external state information
- opcodes that depend on the current PC, memory size or returndata size

DataflowAnalyzer

The DataflowAnalyzer is not an optimizer step itself but is used as a tool by other components. While traversing the AST, it tracks the current value of each variable, as long as that value is a movable expression. It records the variables that are part of the expression that is currently assigned to each other variable. Upon each assignment to a variable `a`, the current stored value of `a` is updated and all stored values of all variables `b` are cleared whenever `a` is part of the currently stored expression for `b`.

At control-flow joins, knowledge about variables is cleared if they have or would be assigned in any of the control-flow paths. For instance, upon entering a `for` loop, all variables are cleared that will be assigned during the body or the post block.

Expression-Scale Simplifications

These simplification passes change expressions and replace them by equivalent and hopefully simpler expressions.

CommonSubexpressionEliminator

This step uses the DataflowAnalyzer and replaces subexpressions that syntactically match the current value of a variable by a reference to that variable. This is an equivalence transform because such subexpressions have to be movable.

All subexpressions that are identifiers themselves are replaced by their current value if the value is an identifier.

The combination of the two rules above allow to compute a local value numbering, which means that if two variables have the same value, one of them will always be unused. The UnusedPruner or the UnusedAssignEliminator will then be able to fully eliminate such variables.

This step is especially efficient if the ExpressionSplitter is run before. If the code is in pseudo-SSA form, the values of variables are available for a longer time and thus we have a higher chance of expressions to be replaceable.

The ExpressionSimplifier will be able to perform better replacements if the CommonSubexpressionEliminator was run right before it.

ExpressionSimplifier

The ExpressionSimplifier uses the DataflowAnalyzer and makes use of a list of equivalence transforms on expressions like `X + 0 -> X` to simplify the code.

It tries to match patterns like `X + 0` on each subexpression. During the matching procedure, it resolves variables to their currently assigned expressions to be able to match more deeply nested patterns even when the code is in pseudo-SSA form.

Some of the patterns like `X - X -> 0` can only be applied as long as the expression `X` is movable, because otherwise it would remove its potential side-effects. Since variable references are always movable, even if their current value might not be, the ExpressionSimplifier is again more powerful in split or pseudo-SSA form.

LiteralRematerialiser

To be documented.

LoadResolver

Optimisation stage that replaces expressions of type `sload(x)` and `mload(x)` by the value currently stored in storage resp. memory, if known.

Works best if the code is in SSA form.

Prerequisites: Disambiguator, ForLoopInitRewriter.

Statement-Scale Simplifications

CircularReferencesPruner

This stage removes functions that call each other but are neither externally referenced nor referenced from the outermost context.

ConditionalSimplifier

The ConditionalSimplifier inserts assignments to condition variables if the value can be determined from the control-flow.

Destroys SSA form.

Currently, this tool is very limited, mostly because we do not yet have support for boolean types. Since conditions only check for expressions being nonzero, we cannot assign a specific value.

Current features:

- `switch` cases: insert `<condition> := <caseLabel>`
- after `if` statement with terminating control-flow, insert `<condition> := 0`

Future features:

- allow replacements by 1
- take termination of user-defined functions into account

Works best with SSA form and if dead code removal has run before.

Prerequisite: Disambiguator.

ConditionalUnsimplifier

Reverse of ConditionalSimplifier.

ControlFlowSimplifier

Simplifies several control-flow structures:

- replace `if` with empty body with `pop(condition)`
- remove empty default `switch` case
- remove empty `switch` case if no default case exists
- replace `switch` with no cases with `pop(expression)`
- turn `switch` with single case into `if`
- replace `switch` with only default case with `pop(expression)` and body
- replace `switch` with const expr with matching case body
- replace `for` with terminating control flow and without other `break`/`continue` by `if`
- remove `leave` at the end of a function.

None of these operations depend on the data flow. The StructuralSimplifier performs similar tasks that do depend on data flow.

The ControlFlowSimplifier does record the presence or absence of `break` and `continue` statements during its traversal.

Prerequisite: Disambiguator, FunctionHoister, ForLoopInitRewriter.

Important: Introduces EVM opcodes and thus can only be used on EVM code for now.

DeadCodeEliminator

This optimization stage removes unreachable code.

Unreachable code is any code within a block which is preceded by a `leave`, `return`, `invalid`, `break`, `continue`, `selfdestruct`, `revert` or by a call to a user-defined function that recurses infinitely.

Function definitions are retained as they might be called by earlier code and thus are considered reachable.

Because variables declared in a `for` loop's init block have their scope extended to the loop body, we require ForLoopInitRewriter to run before this step.

Prerequisites: ForLoopInitRewriter, FunctionHoister, FunctionGrouper.

EqualStoreEliminator

This steps removes `mstore(k, v)` and `sstore(k, v)` calls if there was a previous call to `mstore(k, v)` / `sstore(k, v)`, no other store in between and the values of `k` and `v` did not change.

This simple step is effective if run after the SSATransform and the CommonSubexpressionEliminator, because SSA will make sure that the variables will not change and the CommonSubexpressionEliminator reuses exactly the same variable if the value is known to be the same.

Prerequisites: Disambiguator, ForLoopInitRewriter.

UnusedPruner

This step removes the definitions of all functions that are never referenced.

It also removes declarations of variables that are never referenced. If a declaration assigns a value that is not movable, the expression is retained, but its value is discarded.

All movable expression statements (expressions that are not assigned) are removed.

StructuralSimplifier

This is a general step that performs various kinds of simplifications on a structural level:

- replace `if` statement with empty body by `pop(condition)`
- replace `if` statement with true condition by its body
- remove `if` statement with false condition
- turn `switch` with single case into `if`
- replace `switch` with only default case by `pop(expression)` and body
- replace `switch` with literal expression by matching case body
- replace `for` loop with false condition by its initialization part

This component uses the DataflowAnalyzer.

BlockFlattener

This stage eliminates nested blocks by inserting the statements in the inner block at the appropriate place in the outer block. It depends on the FunctionGrouper and does not flatten the outermost block to keep the form produced by the FunctionGrouper.

```
{
    {
        let x := 2
        {
            let y := 3
            mstore(x, y)
        }
    }
}
```

is transformed to

```
{
    {
        let x := 2
        let y := 3
        mstore(x, y)
    }
}
```

As long as the code is disambiguated, this does not cause a problem because the scopes of variables can only grow.

LoopInvariantCodeMotion

This optimization moves movable SSA variable declarations outside the loop.

Only statements at the top level in a loop's body or post block are considered, i.e variable declarations inside conditional branches will not be moved out of the loop.

ExpressionSplitter and SSATransform should be run upfront to obtain better results.

Prerequisites: Disambiguator, ForLoopInitRewriter, FunctionHoister.

Function-Level Optimizations

FunctionSpecializer

This step specializes the function with its literal arguments.

If a function, say, `function f(a, b) { sstore (a, b) }`, is called with literal arguments, for example, `f(x, 5)`, where `x` is an identifier, it could be specialized by creating a new function `f_1` that takes only one argument, i.e.,

```
function f_1(a_1) {
    let b_1 := 5
    sstore(a_1, b_1)
}
```

Other optimization steps will be able to make more simplifications to the function. The optimization step is mainly useful for functions that would not be inlined.

Prerequisites: Disambiguator, FunctionHoister.

LiteralRematerialiser is recommended as a prerequisite, even though it's not required for correctness.

UnusedFunctionParameterPruner

This step removes unused parameters in a function.

If a parameter is unused, like `c` and `y` in, `function f(a,b,c) -> x, y { x := div(a,b) }`, we remove the parameter and create a new "linking" function as follows:

```
function f(a,b) -> x { x := div(a,b) }
function f2(a,b,c) -> x, y { x := f(a,b) }
```

and replace all references to `f` by `f2`. The inliner should be run afterwards to make sure that all references to `f2` are replaced by `f`.

Prerequisites: Disambiguator, FunctionHoister, LiteralRematerialiser.

The step LiteralRematerialiser is not required for correctness. It helps deal with cases such as: `function f(x) -> y { revert(y, y} }` where the literal `y` will be replaced by its value `0`, allowing us to rewrite the function.

UnusedStoreEliminator

Optimizer component that removes redundant `sstore` and memory store statements. In case of an `sstore`, if all outgoing code paths revert (due to an explicit `revert()`, `invalid()`, or infinite recursion) or lead to another `sstore` for which the optimizer can tell that it will overwrite the first store, the statement will be removed. However, if there is a read operation between the initial `sstore` and the revert, or the overwriting `sstore`, the statement will not be removed. Such read operations include: external calls, user-defined functions with any storage access, and `sload` of a slot that cannot be proven to differ from the slot written by the initial `sstore`.

For example, the following code

```
{
    let c := calldataload(0)
    sstore(c, 1)
    if c {
        sstore(c, 2)
    }
    sstore(c, 3)
}
```

will be transformed into the code below after the UnusedStoreEliminator step is run

```
{
    let c := calldataload(0)
    if c { }
    sstore(c, 3)
}
```

For memory store operations, things are generally simpler, at least in the outermost Yul block as all such statements will be removed if they are never read from in any code path. At function analysis level, however, the approach is similar to `sstore`, as we do not know whether the memory location will be read once we leave the function's scope, so the statement will be removed only if all code paths lead to a memory overwrite.

Best run in SSA form.

Prerequisites: Disambiguator, ForLoopInitRewriter.

EquivalentFunctionCombiner

If two functions are syntactically equivalent, while allowing variable renaming but not any re-ordering, then any reference to one of the functions is replaced by the other.

The actual removal of the function is performed by the UnusedPruner.

Function Inlining

ExpressionInliner

This component of the optimizer performs restricted function inlining by inlining functions that can be inlined inside functional expressions, i.e. functions that:

- return a single value.
- have a body like `r := <functional expression>`.
- neither reference themselves nor `r` in the right hand side.

Furthermore, for all parameters, all of the following need to be true:

- The argument is movable.

- The parameter is either referenced less than twice in the function body, or the argument is rather cheap ("cost" of at most 1, like a constant up to `0xff`).

Example: The function to be inlined has the form of `function f(...) -> r { r := E }` where `E` is an expression that does not reference `r` and all arguments in the function call are movable expressions.

The result of this inlining is always a single expression.

This component can only be used on sources with unique names.

FullInliner

The FullInliner replaces certain calls of certain functions by the function's body. This is not very helpful in most cases, because it just increases the code size but does not have a benefit. Furthermore, code is usually very expensive and we would often rather have shorter code than more efficient code. In some cases, though, inlining a function can have positive effects on subsequent optimizer steps. This is the case if one of the function arguments is a constant, for example.

During inlining, a heuristic is used to tell if the function call should be inlined or not. The current heuristic does not inline into "large" functions unless the called function is tiny. Functions that are only used once are inlined, as well as medium-sized functions, while function calls with constant arguments allow slightly larger functions.

In the future, we may include a backtracking component that, instead of inlining a function right away, only specializes it, which means that a copy of the function is generated where a certain parameter is always replaced by a constant. After that, we can run the optimizer on this specialized function. If it results in heavy gains, the specialized function is kept, otherwise the original function is used instead.

FunctionHoister and ExpressionSplitter are recommended as prerequisites since they make the step more efficient, but are not required for correctness. In particular, function calls with other function calls as arguments are not inlined, but running ExpressionSplitter beforehand ensures that there are no such calls in the input.

Cleanup

The cleanup is performed at the end of the optimizer run. It tries to combine split expressions into deeply nested ones again and also improves the "compilability" for stack machines by eliminating variables as much as possible.

ExpressionJoiner

This is the opposite operation of the ExpressionSplitter. It turns a sequence of variable declarations that have exactly one reference into a complex expression. This stage fully preserves the order of function calls and opcode executions. It does not make use of any information concerning the commutativity of the opcodes; if moving the value of a variable to its place of use would change the order of any function call or opcode execution, the transformation is not performed.

Note that the component will not move the assigned value of a variable assignment or a variable that is referenced more than once.

The snippet `let x := add(0, 2) let y := mul(x, mload(2))` is not transformed, because it would cause the order of the call to the opcodes `add` and `mload` to be swapped - even though this would not make a difference because `add` is movable.

When reordering opcodes like that, variable references and literals are ignored. Because of that, the snippet `let x := add(0, 2) let y := mul(x, 3)` is transformed to `let y := mul(add(0, 2), 3)`, even though the `add` opcode would be executed after the evaluation of the literal `3`.

SSAReverser

This is a tiny step that helps in reversing the effects of the SSATransform if it is combined with the CommonSubexpressionEliminator and the UnusedPruner.

The SSA form we generate is detrimental to code generation because it produces many local variables. It would be better to just reuse existing variables with assignments instead of fresh variable declarations.

The SSATransform rewrites

```
let a := calldataload(0)
mstore(a, 1)
```

to

```
let a_1 := calldataload(0)
let a := a_1
mstore(a_1, 1)
let a_2 := calldataload(0x20)
a := a_2
```

The problem is that instead of `a`, the variable `a_1` is used whenever `a` was referenced. The SSATransform changes statements of this form by just swapping out the declaration and the assignment. The above snippet is turned into

```
let a := calldataload(0)
let a_1 := a
mstore(a_1, 1)
a := calldataload(0x20)
let a_2 := a
```

This is a very simple equivalence transform, but when we now run the CommonSubexpressionEliminator, it will replace all occurrences of `a_1` by `a` (until `a` is re-assigned). The UnusedPruner will then eliminate the variable `a_1` altogether and thus fully reverse the SSATransform.

StackCompressor

One problem that makes code generation for the Ethereum Virtual Machine hard is the fact that there is a hard limit of 16 slots for reaching down the expression stack. This more or less translates to a limit of 16 local variables. The stack compressor takes Yul code and compiles it to EVM bytecode. Whenever the stack difference is too large, it records the function this happened in.

For each function that caused such a problem, the Rematerialiser is called with a special request to aggressively eliminate specific variables sorted by the cost of their values.

On failure, this procedure is repeated multiple times.

Rematerialiser

The rematerialisation stage tries to replace variable references by the expression that was last assigned to the variable. This is of course only beneficial if this expression is comparatively cheap to evaluate. Furthermore, it is only semantically equivalent if the value of the expression did not change between the point of assignment and the point of use. The main benefit of this stage is that it can save stack slots if it leads to a variable being eliminated completely (see below), but it can also save a `DUP` opcode on the EVM if the expression is very cheap.

The Rematerialiser uses the DataflowAnalyzer to track the current values of variables, which are always movable. If the value is very cheap or the variable was explicitly requested to be eliminated, the variable reference is replaced by its current value.

ForLoopConditionOutOfBody

Reverses the transformation of ForLoopConditionIntoBody.

For any movable `c`, it turns

```
for { ... } 1 { ... } {
if iszero(c) { break }
...
}
```

into

```
for { ... } c { ... } {
...
}
```

and it turns

```
for { ... } 1 { ... } {
if c { break }
...
}
```

into

```
for { ... } iszero(c) { ... } {
...
}
```

The LiteralRematerialiser should be run before this step.

3.21.6 Codegen-Based Optimizer Module

Currently, the codegen-based optimizer module provides two optimizations.

The first one, available in the legacy code generator, moves literals to the right side of commutative binary operators, which helps exploit their associativity.

The other one, available in the IR-based code generator, enables the use of unchecked arithmetic when generating code for incrementing the counter variable of certain idiomatic `for` loops. This avoids wasting gas by identifying some conditions that guarantee that the counter variable cannot overflow. This eliminates the need to use a verbose unchecked arithmetic block inside the loop body to increment the counter variable.

Unchecked Loop Increment

Introduced in Solidity `0.8.22`, the overflow check optimization step is concerned with identifying the conditions under which the `for` loop counter can be safely incremented without overflow checks.

This optimization is **only** applied to `for` loops of the general form:

```
for (uint i = X; i < Y; ++i) {
    // variable i is not modified in the loop body
}
```

The condition and the fact that the counter variable is only ever incremented guarantee that it never overflows. The precise requirements for the loop to be eligible for the optimization are as follows:

- The loop condition is a comparison of the form `i < Y`, for a local counter variable `i` (called the "loop counter" hereon) and an expression `Y`.
- The built-in operator < is necessarily used in the loop condition and is the only operator that triggers the optimization. <= and the like are intentionally excluded. Additionally, user-defined operators are **not** eligible.
- The loop expression is a prefix or postfix increment of the counter variable, i.e, `i++` or `++i`.
- The loop counter is a local variable of a built-in integer type.
- The loop counter is **not** modified by the loop body or by the expression used as the loop condition.
- The comparison is performed on the same type as the loop counter, meaning that the type of the right-hand-side expression is implicitly convertible to the type of the counter, such that the latter is not implicitly widened before the comparison.

To clarify the last condition, consider the following example:

```
for (uint8 i = 0; i < uint16(1000); i++) {
    // ...
}
```

In this case, the counter `i` has its type implicitly converted from `uint8` to `uint16` before the comparison and the condition is in fact never false, so the overflow check for the increment cannot be removed.

3.22 Contract Metadata

The Solidity compiler automatically generates a JSON file. The file contains two kinds of information about the compiled contract:

- How to interact with the contract: ABI, and NatSpec documentation.
- How to reproduce the compilation and verify a deployed contract: compiler version, compiler settings, and source files used.

The compiler appends by default the IPFS hash of the metadata file to the end of the runtime bytecode (not necessarily the creation bytecode) of each contract, so that, if published, you can retrieve the file in an authenticated way without having to resort to a centralized data provider. The other available options are the Swarm hash and not appending the metadata hash to the bytecode. These can be configured via the *Standard JSON Interface*.

You have to publish the metadata file to IPFS, Swarm, or another service so that others can access it. You create the file by using the `solc --metadata` command together with the `--output-dir` parameter. Without the parameter, the metadata will be written to standard output. The metadata contains IPFS and Swarm references to the source code, so you have to upload all source files in addition to the metadata file. For IPFS, the hash contained in the CID returned by `ipfs add` (not the direct sha2-256 hash of the file) shall match with the one contained in the bytecode.

The metadata file has the following format. The example below is presented in a human-readable way. Properly formatted metadata should use quotes correctly, reduce whitespace to a minimum, and sort the keys of all objects in alphabetical order to arrive at a canonical formatting. Comments are not permitted and are used here only for explanatory purposes.

```
{
  // Required: Details about the compiler, contents are specific
  // to the language.
  "compiler": {
    // Optional: Hash of the compiler binary which produced this output
    "keccak256": "0x123...",
    // Required for Solidity: Version of the compiler
```

(continues on next page)

(continued from previous page)

```
    "version": "0.8.2+commit.661d1103"
  },
  // Required: Source code language, basically selects a "sub-version"
  // of the specification
  "language": "Solidity",
  // Required: Generated information about the contract.
  "output": {
    // Required: ABI definition of the contract. See "Contract ABI Specification"
    "abi": [/* ... */],
    // Required: NatSpec developer documentation of the contract. See https://docs.
↪soliditylang.org/en/latest/natspec-format.html for details.
    "devdoc": {
      // Contents of the @author NatSpec field of the contract
      "author": "John Doe",
      // Contents of the @dev NatSpec field of the contract
      "details": "Interface of the ERC20 standard as defined in the EIP. See https://
↪eips.ethereum.org/EIPS/eip-20 for details",
      "errors": {
        "MintToZeroAddress()" : {
          "details": "Cannot mint to zero address"
        }
      },
      "events": {
        "Transfer(address,address,uint256)": {
          "details": "Emitted when `value` tokens are moved from one account (`from`)␣
↪toanother (`to`).",
          "params": {
            "from": "The sender address",
            "to": "The receiver address",
            "value": "The token amount"
          }
        }
      },
      "kind": "dev",
      "methods": {
        "transfer(address,uint256)": {
          // Contents of the @dev NatSpec field of the method
          "details": "Returns a boolean value indicating whether the operation succeeded.
↪ Must be called by the token holder address",
          // Contents of the @param NatSpec fields of the method
          "params": {
            "_value": "The amount tokens to be transferred",
            "_to": "The receiver address"
          },
          // Contents of the @return NatSpec field.
          "returns": {
            // Return var name (here "success") if exists. "_0" as key if return var is␣
↪unnamed
            "success": "a boolean value indicating whether the operation succeeded"
          }
        }
      },
```

(continues on next page)

(continued from previous page)

```
      "stateVariables": {
        "owner": {
          // Contents of the @dev NatSpec field of the state variable
          "details": "Must be set during contract creation. Can then only be changed by␣
↪the owner"
        }
      },
      // Contents of the @title NatSpec field of the contract
      "title": "MyERC20: an example ERC20",
      "version": 1 // NatSpec version
    },
    // Required: NatSpec user documentation of the contract. See "NatSpec Format"
    "userdoc": {
      "errors": {
        "ApprovalCallerNotOwnerNorApproved()": [
          {
            "notice": "The caller must own the token or be an approved operator."
          }
        ]
      },
      "events": {
        "Transfer(address,address,uint256)": {
          "notice": "`_value` tokens have been moved from `from` to `to`"
        }
      },
      "kind": "user",
      "methods": {
        "transfer(address,uint256)": {
          "notice": "Transfers `_value` tokens to address `_to`"
        }
      },
      "version": 1 // NatSpec version
    }
  },
  // Required: Compiler settings.
  // Reflects the settings in the JSON input during compilation, except:
  // - Different format: "libraries" field
  // - Added field in metadata.settings: "compilationTarget"
  // - Not in metadata.settings: "stopAfter", "debug.debugInfo", "outputSelection"
  // See the standard JSON input's "settings" field docs for the rest.
  "settings": {
    // Required for Solidity: File path and the name of the contract or library this
    // metadata is created for. This field is not present in the standard JSON input␣
↪settings.
    "compilationTarget": {
      "myDirectory/myFile.sol": "MyContract"
    },
    // Optional (false if omitted): Indicates whether experimental mode has been enabled.
    // Always matches the value of the `experimental` flag in CBOR metadata.
    // Note that experimental mode being enabled does not necessarily mean that any
    // experimental features were actually used, or if they were, that those features
    // affected the bytecode.
```

(continues on next page)

(continued from previous page)

```
    "experimental": true,
    // Required for Solidity: Addresses for libraries used.
    // Note that metadata has a different format for "libraries" field than the standard␣
↪JSON input.
    // metadata format = { "MyLib.sol:MyLib": "0x123123..." }
    // standard JSON input format = { "MyLib.sol": { "MyLib": "0x123123..." } }
    "libraries": {
      "MyLib.sol:MyLib": "0x123123..."
    },
    // ...
    // ...
    // ...
    // The rest of the fields and their defaults same as in std JSON input.
  },
  // Required: Compilation source files/source units, keys are file paths
  "sources": {
    "settable": {
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract settable is owned { uint256 private x = 0; function␣
↪set(uint256 _x) public { if (msg.sender == owner) x = _x; } }",
      // Required: keccak256 hash of the source file
      "keccak256": "0x234..."
    },
    "myDirectory/myFile.sol": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x123...",
      // Optional: SPDX license identifier as given in the source file
      "license": "MIT",
      // Required (unless "content" is used, see above): Sorted URL(s)
      // to the source file, protocol is more or less arbitrary, but an
      // IPFS URL is recommended
      "urls": [ "bzz-raw://7d7a...", "dweb:/ipfs/QmN..." ]
    }
  },
  // Required: The version of the metadata format
  "version": 1
}
```

Warning

Since the bytecode of the resulting contract contains the metadata hash by default, any change to the metadata might result in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode.

Note

The ABI definition above has no fixed order. It can change with compiler versions. Starting from Solidity version 0.5.12, though, the array maintains a certain order.

3.22.1 Encoding of the Metadata Hash in the Bytecode

The compiler currently by default appends the IPFS hash (in CID v0) of the canonical metadata file and the compiler version to the end of the bytecode. Optionally, a Swarm hash instead of the IPFS, or an experimental flag is used. Below are all the possible fields:

```
{
  // Present if "bytecodeHash" was "ipfs" in compiler settings
  "ipfs": "<metadata hash>",
  // Present if "bytecodeHash" was "bzzr1" in compiler settings
  "bzzr1": "<metadata hash>",
  // Previous versions were using "bzzr0" instead of "bzzr1"
  "bzzr0": "<metadata hash>",
  // Present if experimental mode has been enabled either via "--experimental" flag or
  // "settings.experimental" option in Standard JSON
  "experimental": true,
  "solc": "<compiler version>"
}
```

Because we might support other ways to retrieve the metadata file in the future, this information is stored CBOR-encoded. The last two bytes in the bytecode indicate the length of the CBOR encoded information. By looking at this length, the relevant part of the bytecode can be decoded with a CBOR decoder.

Check the Metadata Playground to see it in action.

Whereas release builds of solc use a 3 byte encoding of the version as shown above (one byte each for major, minor and patch version number), pre-release builds will instead use a complete version string including commit hash and build date.

The commandline flag `--no-cbor-metadata` can be used to skip metadata from getting appended at the end of the deployed bytecode. Equivalently, the boolean field `settings.metadata.appendCBOR` in Standard JSON input can be set to false.

Note

The CBOR mapping can also contain other keys, so it is better to fully decode the data by looking at the end of the bytecode for the CBOR length, and to use a proper CBOR parser. Do not rely on it starting with `0xa264` or `0xa2 0x64 'i' 'p' 'f' 's'`.

3.22.2 Usage for Automatic Interface Generation and NatSpec

The metadata is used in the following way: A component that wants to interact with a contract (e.g. a wallet) retrieves the code of the contract. It decodes the CBOR encoded section containing the IPFS/Swarm hash of the metadata file. With that hash, the metadata file is retrieved. That file is JSON-decoded into a structure like above.

The component can then use the ABI to automatically generate a rudimentary user interface for the contract.

Furthermore, the wallet can use the NatSpec user documentation to display a human-readable confirmation message to the user whenever they interact with the contract, together with requesting authorization for the transaction signature.

For additional information, read *Ethereum Natural Language Specification (NatSpec) format*.

3.22.3 Usage for Source Code Verification

If pinned/published, it is possible to retrieve the metadata of the contract from IPFS/Swarm. The metadata file also contains the URLs or the IPFS hashes of the source files, as well as the compilation settings, i.e. everything needed to reproduce a compilation.

With this information it is then possible to verify the source code of a contract by reproducing the compilation, and comparing the bytecode from the compilation with the bytecode of the deployed contract.

This automatically verifies the metadata since its hash is part of the bytecode, as well as the source codes, because their hashes are part of the metadata. Any change in the files or settings would result in a different metadata hash. The metadata here serves as a fingerprint of the whole compilation.

Sourcify makes use of this feature for "full/perfect verification", as well as pinning the files publicly on IPFS to be accessed with the metadata hash.

3.23 Contract ABI Specification

3.23.1 Basic Design

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification. The encoding is not self describing and thus requires a schema in order to decode.

We assume that the interface functions of a contract are strongly typed, known at compilation time and static. We assume that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time. Also, the ABI specification for libraries is *slightly different*.

3.23.2 Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma — no spaces are used.

> **Note**
>
> The return type of a function is not part of this signature. In *Solidity's function overloading* return types are not considered. The reason is to keep function call resolution context-independent. The *JSON description of the ABI* however contains both inputs and outputs.

3.23.3 Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

3.23.4 Types

Note that the library ABIs can take types different than below e.g. for non-storage structs. See *library selectors* for details.

The following elementary types exist:

- `uint<M>`: unsigned integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.

- `int<M>`: two's complement signed integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address`: equivalent to `uint160`, except for the assumed interpretation and language typing. For computing the function selector, `address` is used.
- `uint`, `int`: synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.
- `bool`: equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>`: signed fixed-point decimal number of `M` bits, `8 <= M <= 256`, `M % 8 == 0`, and `0 < N <= 80`, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>`: unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed`: synonyms for `fixed128x18`, `ufixed128x18` respectively. For computing the function selector, `fixed128x18` and `ufixed128x18` have to be used.
- `bytes<M>`: binary type of `M` bytes, `0 < M <= 32`.
- `function`: an address (20 bytes) followed by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists:

- `<type>[M]`: a fixed-length array of `M` elements, `M >= 0`, of the given type.

> **Note**
>
> While this ABI specification can express fixed-length arrays with zero elements, they're not supported by the compiler.

The following non-fixed-size types exist:

- `bytes`: dynamic sized byte sequence.
- `string`: dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]`: a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing them inside parentheses, separated by commas:

- `(T1,T2,...,Tn)`: tuple consisting of the types `T1`, ..., `Tn`, `n >= 0`

It is possible to form tuples of tuples, arrays of tuples and so on. It is also possible to form zero-tuples (where `n == 0`).

Mapping Solidity to ABI types

Solidity supports all the types presented above with the same names with the exception of tuples. On the other hand, some Solidity types are not supported by the ABI. The following table shows on the left column Solidity types that are not part of the ABI, and on the right column the ABI types that represent them.

Solidity	ABI
address payable	`address`
contract	`address`
enum	`uint8`
user defined value types	its underlying value type
struct	`tuple`

Warning

Before version `0.8.0` enums could have more than 256 members and were represented by the smallest integer type just big enough to hold the value of any member.

3.23.5 Design Criteria for the Encoding

The encoding is designed to have the following properties, which are especially useful if some arguments are nested arrays:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or an array element is not interleaved with other data and it is relocatable, i.e. it only uses relative "addresses".

3.23.6 Formal Specification of the Encoding

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition: The following types are called "dynamic":

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k >= 0`
- `(T1,...,Tk)` if `Ti` is dynamic for some `1 <= i <= k`

All other types are called "static".

Definition: `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition: For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `(T1,...,Tk)` for `k >= 0` and any types `T1`, ..., `Tk`

  `enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`

  where `X = (X(1), ..., X(k))` and `head` and `tail` are defined for `Ti` as follows:

  if `Ti` is static:

  `head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

  otherwise, i.e. if `Ti` is dynamic:

  ```
  head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1))
  )) tail(X(i)) = enc(X(i))
  ```

  Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. The value of `head(X(i))` is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  i.e. it is encoded as if it were a tuple with `k` elements of the same type.

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

  `enc(X) = enc(k) enc((X[0], ..., X[k-1]))`

  i.e. it is encoded as if it were a tuple with `k` elements of the same type (resp. an array of static size `k`), prefixed with the number of elements.

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is UTF-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the UTF-8 encoded string, not its number of characters.

- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.

- `address`: as in the `uint160` case

- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-order (left) side with `0xff` bytes for negative `X` and with zero-bytes for non-negative `X` such that the length is 32 bytes.

- `bool`: as in the `uint8` case, where `1` is used for `true` and `0` for `false`

- `fixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `int256`.

- `fixed`: as in the `fixed128x18` case

- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `uint256`.

- `ufixed`: as in the `ufixed128x18` case

- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with trailing zero-bytes to a length of 32 bytes.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

3.23.7 Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values `v_1, ..., v_k` of `f` are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

3.23.8 Examples

Given the contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

Thus, for our `Foo` example, if we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into:

- `0xfce353f6`: the Method ID. This is derived from the signature `bar(bytes3[2])`.
- `0x6162630000000000000000000000000000000000000000000000000000000000`: the first part of the first parameter, a `bytes3` value `"abc"` (left-aligned).
- `0x6465660000000000000000000000000000000000000000000000000000000000`: the second part of the first parameter, a `bytes3` value `"def"` (left-aligned).

In total:

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000000000006465660000000000000000000000000000
```

If we wanted to call `baz` with the parameters `69` and `true`, we would pass 68 bytes total, which can be broken down into:

- `0xcdcd77c0`: the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)`.
- `0x0000000000000000000000000000000000000000000000000000000000000045`: the first parameter, a uint32 value `69` padded to 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter - boolean `true`, padded to 32 bytes

In total:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into:

- `0xa5643bf2`: the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
- `0x0000000000000000000000000000000000000000000000000000000000000060`: the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0`: the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004`: the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.

- `0x6461766500000000000000000000000000000000000000000000000000000000`: the contents of the first argument: the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002`: the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the third entry of the third parameter.

In total:

```
0xa5643bf20000000000000000000000000000000000000000000000000000000000000060000000000000000000000000000000
```

3.23.9 Use of Dynamic Types

A call to a function with the signature `f(uint256,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `keccak("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case): 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity):

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Let us apply the same principle to encode the data for a function with a signature `g(uint256[][],string[])` with values `([[1, 2], [3]], ["one", "two", "three"])` but start from the most atomic parts of the encoding:

First we encode the length and data of the first embedded dynamic array `[1, 2]` of the first root array `[[1, 2], [3]]`:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first array, 2; the elements themselves are 1 and 2)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (second element)

Then we encode the length and data of the second embedded dynamic array `[3]` of the first root array `[[1, 2], [3]]`:

- `0x0000000000000000000000000000000000000000000000000000000000000001` (number of elements in the second array, 1; the element is 3)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (first element)

Then we need to find the offsets `a` and `b` for their respective dynamic arrays `[1, 2]` and `[3]`. To calculate the offsets we can take a look at the encoded data of the first root array `[[1, 2], [3]]` enumerating each line in the encoding:

```
0 - a                                                               - offset of [1, 2]
1 - b                                                               - offset of [3]
2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1, 2]
3 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
4 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
5 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
6 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
```

Offset `a` points to the start of the content of the array `[1, 2]` which is line 2 (64 bytes); thus `a = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `b` points to the start of the content of the array `[3]` which is line 5 (160 bytes); thus `b = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Then we encode the embedded strings of the second root array:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of characters in word `"one"`)
- `0x6f6e650000000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"one"`)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of characters in word `"two"`)
- `0x74776f0000000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"two"`)

- `0x0000000000000000000000000000000000000000000000000000000000000005` (number of characters in word `"three"`)
- `0x7468726565000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"three"`)

In parallel to the first root array, since strings are dynamic elements we need to find their offsets `c`, `d` and `e`:

```
0 - c                                                                - offset for "one"
1 - d                                                                - offset for "two"
2 - e                                                                - offset for "three"
3 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of "one"
5 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of "two"
7 - 0000000000000000000000000000000000000000000000000000000000000005 - count for "three"
8 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of "three
↪"
```

Offset `c` points to the start of the content of the string `"one"` which is line 3 (96 bytes); thus `c` = `0x0000000000000000000000000000000000000000000000000000000000000060`.

Offset `d` points to the start of the content of the string `"two"` which is line 5 (160 bytes); thus `d` = `0x00000000000000000000000000000000000000000000000000000000000000a0`.

Offset `e` points to the start of the content of the string `"three"` which is line 7 (224 bytes); thus `e` = `0x0000000000000000000000000000000000000000000000000000000000000e0`.

Note that the encodings of the embedded elements of the root arrays are not dependent on each other and have the same encodings for a function with a signature `g(string[],uint256[][])`.

Then we encode the length of the first root array:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first root array, 2; the elements themselves are `[1, 2]` and `[3]`)

Then we encode the length of the second root array:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of strings in the second root array, 3; the strings themselves are `"one"`, `"two"` and `"three"`)

Finally we find the offsets `f` and `g` for their respective root dynamic arrays `[[1, 2], [3]]` and `["one", "two", "three"]`, and assemble parts in the correct order:

```
0x2289b18c                                                           - function␣
↪signature
 0 - f                                                               - offset of [[1,␣
↪2], [3]]
 1 - g                                                               - offset of ["one",
↪ "two", "three"]
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [[1,␣
↪2], [3]]
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - offset of [1, 2]
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset of [3]
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1, 2]
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
```

(continues on next page)

(continued from previous page)

```
10 - 0000000000000000000000000000000000000000000000000000000000000003 - count for ["one",
↪ "two", "three"]
11 - 0000000000000000000000000000000000000000000000000000000000000060 - offset for "one"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset for "two"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - offset for "three
↪"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of "one"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of "two"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - count for "three"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset `f` points to the start of the content of the array `[[1, 2], [3]]` which is line 2 (64 bytes); thus `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `g` points to the start of the content of the array `["one", "two", "three"]` which is line 10 (320 bytes); thus `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

3.23.10 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3 (for non-anonymous events) or 4 (for anonymous ones), are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- `address`: the address of the contract (intrinsically provided by Ethereum);
- `topics[0]`: `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo`, it would return `uint256`). This value is only present in `topics[0]` if the event is not declared as `anonymous`;
- `topics[n]`: `abi_encode(EVENT_INDEXED_ARGS[n - 1])` if the event is not declared as `anonymous` or `abi_encode(EVENT_INDEXED_ARGS[n])` if it is (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data`: ABI encoding of `EVENT_NON_INDEXED_ARGS` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_encode` is the ABI encoding function used for returning a series of typed values from a function, as described above).

For all types of length at most 32 bytes, the `EVENT_INDEXED_ARGS` array contains the value directly, padded or sign-extended (for signed integers) to 32 bytes, just as for regular ABI encoding. However, for all "complex" types or types of dynamic length, including all arrays, `string`, `bytes` and structs, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of a special in-place encoded value (see *Encoding of Indexed Event Parameters*), rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this

tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

3.23.11 Errors

In case of a failure inside a contract, the contract can use a special opcode to abort execution and revert all state changes. In addition to these effects, descriptive data can be returned to the caller. This descriptive data is the encoding of an error and its arguments in the same way as data for a function call.

As an example, let us consider the following contract whose `transfer` function always reverts with a custom error of "insufficient balance":

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

The return data would be encoded in the same way as the function call `InsufficientBalance(0, amount)` to the function `InsufficientBalance(uint256,uint256)`, i.e. `0xcf479181`, `uint256(0)`, `uint256(amount)`.

The error selectors `0x00000000` and `0xffffffff` are reserved for future use.

> **Warning**
>
> Never trust error data. The error data by default bubbles up through the chain of external calls, which means that a contract may receive an error not defined in any of the contracts it calls directly. Furthermore, any contract can fake any error by returning data that matches an error signature, even if the error is not defined anywhere.

3.23.12 JSON

The JSON format for a contract's interface is given by an array of function, event and error descriptions. A function description is a JSON object with the fields:

- `type`: `"function"`, `"constructor"`, `"receive"` (the *"receive Ether" function*) or `"fallback"` (the *"default" function*);
- `name`: the name of the function;
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
- `outputs`: an array of objects similar to `inputs`.
- `stateMutability`: a string with one of the following values: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether - the default) and `payable` (function accepts Ether).

Constructor, receive, and fallback never have `name` or `outputs`. Receive and fallback do not have `inputs` either.

Note

Sending non-zero Ether to non-payable function will revert the transaction.

Note

The state mutability `nonpayable` is reflected in Solidity by not specifying a state mutability modifier at all.

An event description is a JSON object with fairly similar fields:

- `type`: always `"event"`
- `name`: the name of the event.
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
  - `indexed`: `true` if the field is part of the log's topics, `false` if it is one of the log's data segments.
- `anonymous`: `true` if the event was declared as `anonymous`.

Errors look as follows:

- `type`: always `"error"`
- `name`: the name of the error.
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).

Note

There can be multiple errors with the same name and even with identical signature in the JSON array; for example, if the errors originate from different files in the smart contract or are referenced from another smart contract. For the ABI, only the name of the error itself is relevant and not where it is defined.

For example,

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;


contract Test {
    constructor() { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    error InsufficientBalance(uint256 available, uint256 required);
```

(continues on next page)

(continued from previous page)

```
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

would result in the JSON:

```
[{
"type":"error",
"inputs": [{"name":"available","type":"uint256"},{"name":"required","type":"uint256"}],
"name":"InsufficientBalance"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Handling tuple types

Despite the fact that names are intentionally not part of the ABI encoding, they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way:

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers `k`. The components of the tuple are then stored in the member `components`, which is of an array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.5 <0.9.0;
pragma abicoder v2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}
```

would result in the JSON:

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
```

(continues on next page)

(continued from previous page)

```
  }
]
```

3.23.13 Strict Encoding Mode

Strict encoding mode is the mode that leads to exactly the same encoding as defined in the formal specification above. This means that offsets have to be as small as possible while still not creating overlaps in the data areas, and thus no gaps are allowed.

Usually, ABI decoders are written in a straightforward way by just following offset pointers, but some decoders might enforce strict mode. The Solidity ABI decoder currently does not enforce strict mode, but the encoder always creates data in strict mode.

3.23.14 Non-standard Packed Mode

Through `abi.encodePacked()`, Solidity supports a non-standard packed mode where:

- types shorter than 32 bytes are concatenated directly, without padding or sign extension
- dynamic types are encoded in-place and without the length.
- array elements are padded, but still encoded in-place

Furthermore, structs as well as nested arrays are not supported.

As an example, the encoding of `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!")` results in:

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                  int16(-1)
      ^^                                bytes1(0x42)
        ^^^^                            uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

More specifically:

- During the encoding, everything is encoded in-place. This means that there is no distinction between head and tail, as in the ABI encoding, and the length of an array is not encoded.
- The direct arguments of `abi.encodePacked` are encoded without padding, as long as they are not arrays (or `string` or `bytes`).
- The encoding of an array is the concatenation of the encoding of its elements **with** padding.
- Dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field.
- The encoding of `string` or `bytes` does not apply padding at the end, unless it is part of an array or struct (then it is padded to a multiple of 32 bytes).

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

If padding is needed, explicit type conversions can be used: `abi.encodePacked(uint16(0x12)) == hex"0012"`.

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector. Since the encoding is ambiguous, there is no decoding function.

Warning

If you use `keccak256(abi.encodePacked(a, b))` and both `a` and `b` are dynamic types, it is easy to craft collisions in the hash value by moving parts of `a` into `b` and vice-versa. More specifically, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. If you use `abi.encodePacked` for signatures, authentication or data integrity, make sure to always use the same types and check that at most one of them is dynamic. Unless there is a compelling reason, `abi.encode` should be preferred.

3.23.15 Encoding of Indexed Event Parameters

Indexed event parameters that are not value types, i.e. arrays and structs are not stored directly but instead a Keccak-256 hash of an encoding is stored. This encoding is defined as follows:

- the encoding of a `bytes` and `string` value is just the string contents without any padding or length prefix.
- the encoding of a struct is the concatenation of the encoding of its members, always padded to a multiple of 32 bytes (even `bytes` and `string`).
- the encoding of an array (both dynamically- and statically-sized) is the concatenation of the encoding of its elements, always padded to a multiple of 32 bytes (even `bytes` and `string`) and without any length prefix

In the above, as usual, a negative number is padded by sign extension and not zero padded. `bytesNN` types are padded on the right while `uintNN` / `intNN` are padded on the left.

Warning

The encoding of a struct is ambiguous if it contains more than one dynamically-sized array. Because of that, always re-check the event data and do not rely on the search result based on the indexed parameters alone.

3.24 Security Considerations

While it is usually quite easy to build software that works as expected, it is much harder to check that nobody can use it in a way that was **not** anticipated.

In Solidity, this is even more important because you can use smart contracts to handle tokens or, possibly, even more valuable things. Furthermore, every execution of a smart contract happens in public and, in addition to that, the source code is often available.

Of course, you always have to consider how much is at stake: You can compare a smart contract with a web service that is open to the public (and thus, also to malicious actors) and perhaps even open-source. If you only store your grocery list on that web service, you might not have to take too much care, but if you manage your bank account using that web service, you should be more careful.

This section will list some pitfalls and general security recommendations but can, of course, never be complete. Also, keep in mind that even if your smart contract code is bug-free, the compiler or the platform itself might have a bug. A list of some publicly known security-relevant bugs of the compiler can be found in the *list of known bugs*, which is also machine-readable. Note that there is a Bug Bounty Program that covers the code generator of the Solidity compiler.

As always, with open-source documentation, please help us extend this section (especially, some examples would not hurt)!

NOTE: In addition to the list below, you can find more security recommendations and best practices in Guy Lando's knowledge list and the Consensys GitHub repo.

3.24.1 Pitfalls

Private Information and Randomness

Everything you use in a smart contract is publicly visible, even local variables and state variables marked `private`.

Using random numbers in smart contracts is quite tricky if you do not want block builders to be able to cheat.

Reentrancy

Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed. To give an example, the following code contains a bug (it is just a snippet and not a complete contract):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
    // This will report a warning (deprecation)
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

The problem is not too serious here because of the limited gas as part of `send`, but it still exposes a weakness: Ether transfer can always include code execution, so the recipient could be a contract that calls back into `withdraw`. This would let it get multiple refunds and, basically, retrieve all the Ether in the contract. In particular, the following contract will allow an attacker to refund multiple times as it uses `call` which does not limit the amount of gas that is forwarded by default:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

To avoid reentrancy, you can use the Checks-Effects-Interactions pattern as demonstrated below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;
```

(continues on next page)

(continued from previous page)

```
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        (bool success, ) = payable(msg.sender).call{value: share}("");
        require(success);
    }
}
```

The Checks-Effects-Interactions pattern ensures that all code paths through a contract complete all required checks of the supplied parameters before modifying the contract's state (Checks); only then it makes any changes to the state (Effects); it may make calls to functions in other contracts *after* all planned state changes have been written to storage (Interactions). This is a common foolproof way to prevent *reentrancy attacks*, where an externally called malicious contract can double-spend an allowance, double-withdraw a balance, among other things, by using logic that calls back into the original contract before it has finalized its transaction.

Note that reentrancy is not only an effect of Ether transfer but of any function call on another contract. Furthermore, you also have to take multi-contract situations into account. A called contract could modify the state of another contract you depend on.

Gas Limit and Loops

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully: Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. This may not apply to `view` functions that are only executed to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those. Please be explicit about such cases in the documentation of your contracts.

Sending and Receiving Ether

- Neither contracts nor "externally-owned accounts" are currently able to prevent someone from sending them Ether. Contracts can react on and reject a regular transfer, but there are ways to move Ether without creating a message call. One way is to simply "mine to" the contract address and the second way is using `selfdestruct(x)`.
- If a contract receives Ether (without a function being called), either the *receive Ether* or the *fallback* function is executed. If it does not have a `receive` nor a `fallback` function, the Ether will be rejected (by throwing an exception). During the execution of one of these functions, the contract can only rely on the "gas stipend" it is passed (2300 gas) being available to it at that time. This stipend is not enough to modify storage (do not take this for granted though, the stipend might change with future hard forks). To be sure that your contract can receive Ether in that way, check the gas requirements of the receive and fallback functions (for example in the "details" section in Remix).
- There is a way to forward more gas to the receiving contract using `addr.call{value: x}("")`. This is essentially the same as `addr.transfer(x)`, only that it forwards all remaining gas, subject to additional limits imposed by some EVM versions (such as the 63/64th rule introduced by `tangerineWhistle`), and opens up the ability for the recipient to perform more expensive actions (and it returns a failure code instead of automatically propagating the error). This might include calling back into the sending contract or other state changes you might not have thought of. So it allows for great flexibility for honest users but also for malicious actors.
- Use the most precise units to represent the Wei amount as possible, as you lose any that is rounded due to a lack of precision.

- If you want to send Ether using `address.transfer`, there are certain details to be aware of:
  1. If the recipient is a contract, it causes its receive or fallback function to be executed which can, in turn, call back the sending contract.
  2. Sending Ether can fail due to the call depth going above 1024. Since the caller is in total control of the call depth, they can force the transfer to fail; take this possibility into account or use `send` and make sure to always check its return value. Better yet, write your contract using a pattern where the recipient can withdraw Ether instead.
  3. Sending Ether can also fail because the execution of the recipient contract requires more than the allotted amount of gas (explicitly by using *require*, *assert*, *revert* or because the operation is too expensive) - it "runs out of gas" (OOG). If you use `transfer` or `send` with a return value check, this might provide a means for the recipient to block progress in the sending contract. Again, the best practice here is to use a *"withdraw" pattern instead of a "send" pattern*.

Call Stack Depth

External function calls can fail at any time because they exceed the maximum call stack size limit of 1024. In such situations, Solidity throws an exception. Malicious actors might be able to force the call stack to a high value before they interact with your contract. Note that, since Tangerine Whistle hardfork, the 63/64 rule makes call stack depth attack impractical. Also note that the call stack and the expression stack are unrelated, even though both have a size limit of 1024 stack slots.

Note that `.send()` does **not** throw an exception if the call stack is depleted but rather returns `false` in that case. The low-level functions `.call()`, `.delegatecall()` and `.staticcall()` behave in the same way.

Authorized Proxies

If your contract can act as a proxy, i.e. if it can call arbitrary contracts with user-supplied data, then the user can essentially assume the identity of the proxy contract. Even if you have other protective measures in place, it is best to build your contract system such that the proxy does not have any permissions (not even for itself). If needed, you can accomplish that using a second proxy:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ProxyWithMoreFunctionality {
    PermissionlessProxy proxy;

    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return proxy.callOther(addr, payload);
    }
    // Other functions and other functionality
}

// This is the full contract, it has no other functionality and
// requires no privileges to work.
contract PermissionlessProxy {
    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return addr.call(payload);
    }
}
```

tx.origin

Never use `tx.origin` for authorization. Let's say you have a wallet contract like this:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        // THE BUG IS RIGHT HERE, you must use msg.sender instead of tx.origin
        require(tx.origin == owner);
        // This will report a warning (deprecation)
        dest.transfer(amount);
    }
}
```

Now someone tricks you into sending Ether to the address of this attack wallet:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

If your wallet had checked `msg.sender` for authorization, it would get the address of the attack wallet, instead of the owner's address. But by checking `tx.origin`, it gets the original address that kicked off the transaction, which is still the owner's address. The attack wallet instantly drains all your funds.

Two's Complement / Underflows / Overflows

As in many programming languages, Solidity's integer types are not actually integers. They resemble integers when the values are small, but cannot represent arbitrarily large numbers.

The following code causes an overflow because the result of the addition is too large to be stored in the type `uint8`:

```
uint8 x = 255;
uint8 y = 1;
```

(continues on next page)

(continued from previous page)

```
return x + y;
```

Solidity has two modes in which it deals with these overflows: Checked and Unchecked or "wrapping" mode.

The default checked mode will detect overflows and cause a failing assertion. You can disable this check using `unchecked { ... }`, causing the overflow to be silently ignored. The above code would return `0` if wrapped in `unchecked { ... }`.

Even in checked mode, do not assume you are protected from overflow bugs. In this mode, overflows will always revert. If it is not possible to avoid the overflow, this can lead to a smart contract being stuck in a certain state.

In general, read about the limits of two's complement representation, which even has some more special edge cases for signed numbers.

Try to use `require` to limit the size of inputs to a reasonable range and use the *SMT checker* to find potential overflows.

Clearing Mappings

The Solidity type `mapping` (see *Mapping Types*) is a storage-only key-value data structure that does not keep track of the keys that were assigned a non-zero value. Because of that, cleaning a mapping without extra information about the written keys is not possible. If a `mapping` is used as the base type of a dynamic storage array, deleting or popping the array will have no effect over the `mapping` elements. The same happens, for example, if a `mapping` is used as the type of a member field of a `struct` that is the base type of a dynamic storage array. The `mapping` is also ignored in assignments of structs or arrays containing a `mapping`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Map {
    mapping(uint => uint)[] array;

    function allocate(uint newMaps) public {
        for (uint i = 0; i < newMaps; i++)
            array.push();
    }

    function writeMap(uint map, uint key, uint value) public {
        array[map][key] = value;
    }

    function readMap(uint map, uint key) public view returns (uint) {
        return array[map][key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

Consider the example above and the following sequence of calls: `allocate(10)`, `writeMap(4, 128, 256)`. At this point, calling `readMap(4, 128)` returns 256. If we call `eraseMaps`, the length of the state variable `array` is zeroed, but since its `mapping` elements cannot be zeroed, their information stays alive in the contract's storage. After deleting `array`, calling `allocate(5)` allows us to access `array[4]` again, and calling `readMap(4, 128)` returns 256 even without another call to `writeMap`.

If your `mapping` information must be deleted, consider using a library similar to iterable mapping, allowing you to traverse the keys and delete their values in the appropriate `mapping`.

Internal Function Pointers in Upgradeable Contracts

Updating the code of your contract may *invalidate the values of variables of internal function types*. Consider such values ephemeral and avoid storing them in state variables. If you do, you must ensure that they never persist across code updates and are never used by other contracts having access to the same storage space as a result of a delegatecall or account abstraction.

Minor Details

- Types that do not occupy the full 32 bytes might contain "dirty higher order bits". This is especially important if you access `msg.data` - it poses a malleability risk: You can craft transactions that call a function `f(uint8 x)` with a raw byte argument of `0xff000001` and with `0x00000001`. Both are fed to the contract and both will look like the number `1` as far as `x` is concerned, but `msg.data` will be different, so if you use `keccak256(msg.data)` for anything, you will get different results.

3.24.2 Recommendations

Take Warnings Seriously

If the compiler warns you about something, you should change it. Even if you do not think that this particular warning has security implications, there might be another issue buried beneath it. Any compiler warning we issue can be silenced by slight changes to the code.

Always use the latest version of the compiler to be notified about all recently introduced warnings.

Messages of type `info`, issued by the compiler, are not dangerous and simply represent extra suggestions and optional information that the compiler thinks might be useful to the user.

Restrict the Amount of Ether

Restrict the amount of Ether (or other tokens) that can be stored in a smart contract. If your source code, the compiler or the platform has a bug, these funds may be lost. If you want to limit your loss, limit the amount of Ether.

Keep it Small and Modular

Keep your contracts small and easily understandable. Single out unrelated functionality in other contracts or into libraries. General recommendations about the source code quality of course apply: Limit the amount of local variables, the length of functions and so on. Document your functions so that others can see what your intention was and whether it is different than what the code does.

Use the Checks-Effects-Interactions Pattern

Most functions will first perform some checks and they should be done first (who called the function, are the arguments in range, did they send enough Ether, does the person have tokens, etc.).

As the second step, if all checks passed, effects to the state variables of the current contract should be made. Interaction with other contracts should be the very last step in any function.

Early contracts delayed some effects and waited for external function calls to return in a non-error state. This is often a serious mistake because of the reentrancy problem explained above.

Note that, also, calls to known contracts might in turn cause calls to unknown contracts, so it is probably better to just always apply this pattern.

Include a Fail-Safe Mode

While making your system fully decentralized will remove any intermediary, it might be a good idea, especially for new code, to include some kind of fail-safe mechanism:

You can add a function in your smart contract that performs some self-checks like "Has any Ether leaked?", "Is the sum of the tokens equal to the balance of the contract?" or similar things. Keep in mind that you cannot use too much gas for that, so help through off-chain computations might be needed there.

If the self-check fails, the contract automatically switches into some kind of "failsafe" mode, which, for example, disables most of the features, hands over control to a fixed and trusted third party or just converts the contract into a simple "give me back my Ether" contract.

Ask for Peer Review

The more people examine a piece of code, the more issues are found. Asking people to review your code also helps as a cross-check to find out whether your code is easy to understand - a very important criterion for good smart contracts.

3.25 List of Known Bugs

Below, you can find a JSON-formatted list of some of the known security-relevant bugs in the Solidity compiler. The file itself is hosted in the GitHub repository. The list stretches back as far as version 0.3.0, bugs known to be present only in versions preceding that are not listed.

There is another file called bugs_by_version.json, which can be used to check which bugs affect a specific version of the compiler.

Contract source verification tools and also other tools interacting with contracts should consult this list according to the following criteria:

- It is mildly suspicious if a contract was compiled with a nightly compiler version instead of a released version. This list does not keep track of unreleased or nightly versions.
- It is also mildly suspicious if a contract was compiled with a version that was not the most recent at the time the contract was created. For contracts created from other contracts, you have to follow the creation chain back to a transaction and use the date of that transaction as creation date.
- It is highly suspicious if a contract was compiled with a compiler that contains a known bug and the contract was created at a time where a newer compiler version containing a fix was already released.

The JSON file of known bugs below is an array of objects, one for each bug, with the following keys:

uid
: Unique identifier given to the bug in the form of `SOL-<year>-<number>`. It is possible that multiple entries exists with the same uid. This means multiple version ranges are affected by the same bug.

name
: Unique name given to the bug

summary
: Short description of the bug

description
: Detailed description of the bug

link
: URL of a website with more detailed information, optional

introduced
: The first published compiler version that contained the bug, optional

fixed
The first published compiler version that did not contain the bug anymore

publish
The date at which the bug became known publicly, optional

severity
Severity of the bug: very low, low, medium, high. Takes into account discoverability in contract tests, likelihood of occurrence and potential damage by exploits.

conditions
Conditions that have to be met to trigger the bug. The following keys can be used: `optimizer`, Boolean value which means that the optimizer has to be switched on to enable the bug. `evmVersion`, a string that indicates which EVM version compiler settings trigger the bug. The string can contain comparison operators. For example, `">=constantinople"` means that the bug is present when the EVM version is set to `constantinople` or later. If no conditions are given, assume that the bug is present.

check
This field contains different checks that report whether the smart contract contains the bug or not. The first type of check are JavaScript regular expressions that are to be matched against the source code ("source-regex") if the bug is present. If there is no match, then the bug is very likely not present. If there is a match, the bug might be present. For improved accuracy, the checks should be applied to the source code after stripping comments. The second type of check are patterns to be checked on the compact AST of the Solidity program ("ast-compact-json-path"). The specified search query is a JsonPath expression. If at least one path of the Solidity AST matches the query, the bug is likely present.

```
[
    {
        "uid": "SOL-2026-3",
        "name": "InheritanceOrderReversalOnStorageEndWarning",
        "summary": "Emitting a warning about storage base location being too close to␣
↪the storage end unintentionally reversed the ``linearizedBaseContracts`` annotation,␣
↪possibly leading to miscompilation due to reversing the order in which inheritance is␣
↪resolved.",
        "description": "When the compiler detects that a custom layout specifier puts␣
↪contract's static storage area too close to the end of the address space, it emits a␣
↪warning. To make the warning more useful, the compiler tries to point at the last␣
↪storage variable in that area. For this reason it walks the linearized inheritance␣
↪hierarchy in reverse (from the least to the most derived). The list is calculated once␣
↪and stored in an AST annotation called ``linearizedBaseContracts``. The direct cause␣
↪of the bug was the fact that the code that reverses the list was doing it in place␣
↪rather than on a copy, modifying the annotation. This effectively reversed the order␣
↪of base contracts seen by any component that runs after layout checks: later phases of␣
↪analysis, AST export, code generator, SMTChecker, etc. The observable effect was a␣
↪reversed order of state variable initialization, constructor invocation, virtual␣
↪function/modifier resolution, leading either to miscompilations or internal compiler␣
↪errors, depending on the specific usage. Since the source of the bug was in the␣
↪analysis stage, it was independent of the codegen pipeline or optimizer settings. The␣
↪main condition necessary to trigger the bug was the presence of the warning in the␣
↪output. The other is presence of language constructs whose evaluation depends on the␣
↪inheritance order. While potential effects are very serious, this requirement excludes␣
↪the vast majority of contracts as intentionally placing the storage variables in the␣
↪last 2**64 slots is highly discouraged, which was actually the main reason for adding␣
↪this warning.",
        "link": "https://blog.soliditylang.org/2026/07/09/inheritance-order-reversal-on-
```

(continues on next page)

(continued from previous page)

```
↪storage-end-warning-bug/",
        "introduced": "0.8.29",
        "fixed": "0.8.36",
        "severity": "medium"
    },
    {
        "uid": "SOL-2026-2",
        "name": "UnsoundSpillInMutualRecursion",
        "summary": "Local variables of a function involved in mutual recursion may␣
↪spuriously be moved to fixed memory offsets and overwritten across recursive calls.",
        "description": "To work around the 16-slot stack access limit of the EVM, the IR-
↪based code generator can move local variables of stack-too-deep functions to fixed␣
↪memory offsets. This relocation is unsound for recursive functions: a fixed offset␣
↪would be shared by all activations of the function, so a recursive call would␣
↪overwrite the caller's value. The stack limit evader therefore must not relocate␣
↪variables of functions that are part of a recursive call chain. To this end, the call␣
↪graph was searched for cycles using a path-based depth-first search that, once a␣
↪function had been fully explored and popped from the search path, short-circuited on␣
↪it on any later visit. As a result, a function shared between several intersecting␣
↪cycles could be reached first through a path that did not yet close a cycle through it,
↪ get marked as finished, and then be skipped when a later path would have revealed␣
↪that it does lie on a cycle. Such a function was misclassified as non-recursive. When␣
↪a misclassified function was complex enough for the stack limit evader to relocate␣
↪some of its variables, those variables were moved to fixed memory offsets and silently␣
↪corrupted on recursion, producing wrong results rather than a compile-time error.␣
↪Triggering the bug requires the IR pipeline, a set of mutually recursive functions␣
↪whose call graph contains intersecting cycles, at least one of the functions in an␣
↪undetected part of a cycle being complex enough to require relocation to memory, and␣
↪an unfortunate processing order of the functions (which depends on the hashes of their␣
↪Yul names). It is independent of whether the optimizer is enabled.",
        "link": "https://blog.soliditylang.org/2026/07/09/unsound-spill-in-mutual-
↪recursion-bug/",
        "introduced": "0.7.2",
        "fixed": "0.8.36",
        "severity": "medium",
        "conditions": {
            "viaIR": true
        }
    },
    {
        "uid": "SOL-2026-1",
        "name": "TransientStorageClearingHelperCollision",
        "summary": "Clearing both storage and transient storage variables in the same␣
↪contract may result in only one of these locations being cleared.",
        "description": "The IR-based code generator provides a set of Yul helper␣
↪functions for basic operations, such as clearing, copying, encoding or type␣
↪conversions. Not all functions are used by every contract. The codegen appends them to␣
↪the generated sources individually, only when an operation that would invoke one of␣
↪them is encountered. Utility functions are often specialized for different types and␣
↪locations. Since Yul does not support generic functions, specialization is done by␣
↪generating multiple versions of the same function, with the information distinguishing␣
↪the variants embedded in their names. However, if not all the necessary bits of␣
```

(continues on next page)

(continued from previous page)

```
↪distinguishing information are properly accounted for, two helpers may end up with the␣
↪same name, causing a collision. In this situation the codegen includes only one of␣
↪them, with calls to both variants invoking it. This happened with the ``set_to_zero``␣
↪helper used when an area of transient or persistent storage needs to be cleared. The␣
↪helper name was missing the location information, which resulted in a collision␣
↪between the persistent and transient storage variants for the same type. This meant␣
↪that contracts clearing both locations would actually clear only one, leaving the␣
↪other untouched. Which location ended up being cleared depended on the order in which␣
↪the code generator processed the input. The necessary condition to trigger the bug was␣
↪the use of ``delete`` operator on a transient storage variable. This was due to value␣
↪types being the only types supported in transient storage and ``delete`` being the␣
↪only operation invoking the helper allowed on such types. The other necessary␣
↪condition was clearing of persistent storage and in this case the range of affected␣
↪operations was wider: operator ``delete``, array ``pop()`` or assignment that resulted␣
↪in a longer array being overwritten with a shorter one. The cleared variable itself␣
↪also did not necessarily have to be of the same type. It was enough that a matching␣
↪value type was nested in it. It also did not always have to be the exact same value␣
↪type - clearing operations on reference types are usually performed at slot␣
↪granularity, treating every slot as ``uint256`` rather than clearing every value␣
↪packed into it individually. To trigger the bug both operations had to be present␣
↪within the same piece of bytecode. Independent contracts, not related through␣
↪inheritance, would not affect each other this way. The presence of one operation only␣
↪in creation code and the other only in deployed code would not trigger the bug either.
↪",
        "link": "https://blog.soliditylang.org/2026/02/18/transient-storage-clearing-
↪helper-collision-bug/",
        "introduced": "0.8.28",
        "fixed": "0.8.34",
        "severity": "high",
        "conditions": {
            "viaIR": true,
            "evmVersion": ">=cancun"
        }
    },
    {
        "uid": "SOL-2025-1",
        "name": "LostStorageArrayWriteOnSlotOverflow",
        "summary": "Operations that involve clearing or copying from arrays that␣
↪straddle the end of storage could result in silent data retention.",
        "description": "Solidity makes it possible to define variables that extend past␣
↪the last (2**256-th) slot of storage, which results in wrap-around back to slot zero.␣
↪Since EVM uses 256-bit integer arithmetic, most operations on such variables just work.
↪ The only situation which requires special attention is iteration against absolute␣
↪slot addresses: the invariant that the last slot belonging to a variable has the␣
↪highest address does not hold. When implemented incorrectly, a loop over an array will␣
↪immediately terminate if the container spans the end of storage - due to the initial␣
↪position already being greater than the end position. This affected storage array␣
↪clearing loops generated by both evmasm and IR pipelines. Additionally, (only in the␣
↪evmasm pipeline) copying operations whose source was an array straddling the end of␣
↪storage were also affected. At the language level, the buggy code would be generated␣
↪for array assignment, array initialization, delete operator, <array>.pop() and <array>.
↪push(). Note that a clearing loop is inserted by the compiler not only for invocations␣
```

(continues on next page)

(continued from previous page)

```
↪of the delete operator, but also to zero storage when overwriting a longer array with␣
↪a shorter one, popping an element or even pushing an empty element to a dynamic array.␣
↪Since clearing is a separate loop, it is possible for the bug to only affect it and␣
↪not the copy operation it follows (which is always the case in the IR pipeline). The␣
↪bug is extremely unlikely to be triggered accidentally due to the probabilistic␣
↪impossibility of a short dynamic array being allocated right at the storage boundary.␣
↪On the other hand, scenarios in which a user may place a static array there␣
↪intentionally do not seem realistic and are limited to unusual layouts, in which a␣
↪contract does not place any storage variables at slot zero (otherwise they would␣
↪overlap the array).",
        "link": "https://blog.soliditylang.org/2025/12/18/lost-storage-array-write-on-
↪slot-overflow-bug/",
        "introduced": "0.1.0",
        "fixed": "0.8.32",
        "severity": "low"
    },
    {
        "uid": "SOL-2023-3",
        "name": "VerbatimInvalidDeduplication",
        "summary": "All ``verbatim`` blocks are considered identical by deduplicator and␣
↪can incorrectly be unified when surrounded by identical opcodes.",
        "description": "The block deduplicator is a step of the opcode-based optimizer␣
↪which identifies equivalent assembly blocks and merges them into a single one. However,
↪ when blocks contained ``verbatim``, their comparison was performed incorrectly,␣
↪leading to the collapse of assembly blocks which are identical except for the contents␣
↪of the ``verbatim`` items. Since ``verbatim`` is only available in Yul, compilation of␣
↪Solidity sources is not affected.",
        "link": "https://blog.soliditylang.org/2023/11/08/verbatim-invalid-deduplication-
↪bug/",
        "introduced": "0.8.5",
        "fixed": "0.8.23",
        "severity": "low"
    },
    {
        "uid": "SOL-2023-2",
        "name": "FullInlinerNonExpressionSplitArgumentEvaluationOrder",
        "summary": "Optimizer sequences containing FullInliner do not preserve the␣
↪evaluation order of arguments of inlined function calls in code that is not in␣
↪expression-split form.",
        "description": "Function call arguments in Yul are evaluated right to left. This␣
↪order matters when the argument expressions have side-effects, and changing it may␣
↪change contract behavior. FullInliner is an optimizer step that can replace a function␣
↪call with the body of that function. The transformation involves assigning argument␣
↪expressions to temporary variables, which imposes an explicit evaluation order.␣
↪FullInliner was written with the assumption that this order does not necessarily have␣
↪to match usual argument evaluation order because the argument expressions have no side-
↪effects. In most circumstances this assumption is true because the default␣
↪optimization step sequence contains the ExpressionSplitter step. ExpressionSplitter␣
↪ensures that the code is in *expression-split form*, which means that function calls␣
↪cannot appear nested inside expressions, and all function call arguments have to be␣
↪variables. The assumption is, however, not guaranteed to be true in general. Version 0.
↪6.7 introduced a setting allowing users to specify an arbitrary optimization step␣
```

(continues on next page)

(continued from previous page)

```
↪sequence, making it possible for the FullInliner to actually encounter argument␣
↪expressions with side-effects, which can result in behavior differences between␣
↪optimized and unoptimized bytecode. Contracts compiled without optimization or with␣
↪the default optimization sequence are not affected. To trigger the bug the user has to␣
↪explicitly choose compiler settings that contain a sequence with FullInliner step not␣
↪preceded by ExpressionSplitter.",
        "link": "https://blog.soliditylang.org/2023/07/19/full-inliner-non-expression-
↪split-argument-evaluation-order-bug/",
        "introduced": "0.6.7",
        "fixed": "0.8.21",
        "severity": "low",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2023-1",
        "name": "MissingSideEffectsOnSelectorAccess",
        "summary": "Accessing the ``.selector`` member on complex expressions leaves the␣
↪expression unevaluated in the legacy code generation.",
        "description": "When accessing the ``.selector`` member on an expression with␣
↪side-effects, like an assignment, a function call or a conditional, the expression␣
↪would not be evaluated in the legacy code generation. This would happen in expressions␣
↪where the functions used in the expression were all known at compilation time,␣
↪regardless of whether the whole expression could be evaluated at compilation time or␣
↪not. Note that the code generated by the IR pipeline was unaffected and would behave␣
↪as expected.",
        "link": "https://blog.soliditylang.org/2023/07/19/missing-side-effects-on-
↪selector-access-bug/",
        "introduced": "0.6.2",
        "fixed": "0.8.21",
        "severity": "low",
        "conditions": {
            "viaIR": false
        }
    },
    {
        "uid": "SOL-2022-7",
        "name": "StorageWriteRemovalBeforeConditionalTermination",
        "summary": "Calling functions that conditionally terminate the external EVM call␣
↪using the assembly statements ``return(...)`` or ``stop()`` may result in incorrect␣
↪removals of prior storage writes.",
        "description": "A call to a Yul function that conditionally terminates the␣
↪external EVM call could result in prior storage writes being incorrectly removed by␣
↪the Yul optimizer. This used to happen in cases in which it would have been valid to␣
↪remove the store, if the Yul function in question never actually terminated the␣
↪external call, and the control flow always returned back to the caller instead.␣
↪Conditional termination within the same Yul block instead of within a called function␣
↪was not affected. In Solidity with optimized via-IR code generation, any storage write␣
↪before a function conditionally calling ``return(...)`` or ``stop()`` in inline␣
↪assembly, may have been incorrectly removed, whenever it would have been valid to␣
↪remove the write without the ``return(...)`` or ``stop()``. In optimized legacy code␣
```

(continues on next page)

(continued from previous page)

```
↪generation, only inline assembly that did not refer to any Solidity variables and that␣
↪involved conditionally-terminating user-defined assembly functions could be affected.",
        "link": "https://blog.soliditylang.org/2022/09/08/storage-write-removal-before-
↪conditional-termination/",
        "introduced": "0.8.13",
        "fixed": "0.8.17",
        "severity": "medium/high",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-6",
        "name": "AbiReencodingHeadOverflowWithStaticArrayCleanup",
        "summary": "ABI-encoding a tuple with a statically-sized calldata array in the␣
↪last component would corrupt 32 leading bytes of its first dynamically encoded␣
↪component.",
        "description": "When ABI-encoding a statically-sized calldata array, the␣
↪compiler always pads the data area to a multiple of 32-bytes and ensures that the␣
↪padding bytes are zeroed. In some cases, this cleanup used to be performed by always␣
↪writing exactly 32 bytes, regardless of how many needed to be zeroed. This was done␣
↪with the assumption that the data that would eventually occupy the area past the end␣
↪of the array had not yet been written, because the encoder processes tuple components␣
↪in the order they were given. While this assumption is mostly true, there is an␣
↪important corner case: dynamically encoded tuple components are stored separately from␣
↪the statically-sized ones in an area called the *tail* of the encoding and the tail␣
↪immediately follows the *head*, which is where the statically-sized components are␣
↪placed. The aforementioned cleanup, if performed for the last component of the head␣
↪would cross into the tail and overwrite up to 32 bytes of the first component stored␣
↪there with zeros. The only array type for which the cleanup could actually result in␣
↪an overwrite were arrays with ``uint256`` or ``bytes32`` as the base element type and␣
↪in this case the size of the corrupted area was always exactly 32 bytes. The problem␣
↪affected tuples at any nesting level. This included also structs, which are encoded as␣
↪tuples in the ABI. Note also that lists of parameters and return values of functions,␣
↪events and errors are encoded as tuples.",
        "link": "https://blog.soliditylang.org/2022/08/08/calldata-tuple-reencoding-head-
↪overflow-bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.16",
        "severity": "medium",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2022-5",
        "name": "DirtyBytesArrayToStorage",
        "summary": "Copying ``bytes`` arrays from memory or calldata to storage may␣
↪result in dirty storage values.",
        "description": "Copying ``bytes`` arrays from memory or calldata to storage is␣
↪done in chunks of 32 bytes even if the length is not a multiple of 32. Thereby, extra␣
↪bytes past the end of the array may be copied from calldata or memory to storage.␣
```

(continues on next page)

(continued from previous page)

```
↪These dirty bytes may then become observable after a ``.push()`` without arguments to␣
↪the bytes array in storage, i.e. such a push will not result in a zero value at the␣
↪end of the array as expected. This bug only affects the legacy code generation␣
↪pipeline, the new code generation pipeline via IR is not affected.",
        "link": "https://blog.soliditylang.org/2022/06/15/dirty-bytes-array-to-storage-
↪bug/",
        "introduced": "0.0.1",
        "fixed": "0.8.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2022-4",
        "name": "InlineAssemblyMemorySideEffects",
        "summary": "The Yul optimizer may incorrectly remove memory writes from inline␣
↪assembly blocks, that do not access solidity variables.",
        "description": "The Yul optimizer considers all memory writes in the outermost␣
↪Yul block that are never read from as unused and removes them. This is valid when that␣
↪Yul block is the entire Yul program, which is always the case for the Yul code␣
↪generated by the new via-IR pipeline. Inline assembly blocks are never optimized in␣
↪isolation when using that pipeline. Instead they are optimized as a part of the whole␣
↪Yul input. However, the legacy code generation pipeline (which is still the default)␣
↪runs the Yul optimizer individually on an inline assembly block if the block does not␣
↪refer to any local variables defined in the surrounding Solidity code. Consequently,␣
↪memory writes in such inline assembly blocks are removed as well, if the written␣
↪memory is never read from in the same assembly block, even if the written memory is␣
↪accessed later, for example by a subsequent inline assembly block.",
        "link": "https://blog.soliditylang.org/2022/06/15/inline-assembly-memory-side-
↪effects-bug/",
        "introduced": "0.8.13",
        "fixed": "0.8.15",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-3",
        "name": "DataLocationChangeInInternalOverride",
        "summary": "It was possible to change the data location of the parameters or␣
↪return variables from ``calldata`` to ``memory`` and vice-versa while overriding␣
↪internal and public functions. This caused invalid code to be generated when calling␣
↪such a function internally through virtual function calls.",
        "description": "When calling external functions, it is irrelevant if the data␣
↪location of the parameters is ``calldata`` or ``memory``, the encoding of the data␣
↪does not change. Because of that, changing the data location when overriding external␣
↪functions is allowed. The compiler incorrectly also allowed a change in the data␣
↪location for overriding public and internal functions. Since public functions can be␣
↪called internally as well as externally, this causes invalid code to be generated when␣
↪such an incorrectly overridden function is called internally through the base contract.
↪ The caller provides a memory pointer, but the called function interprets it as a␣
↪calldata pointer or vice-versa.",
        "link": "https://blog.soliditylang.org/2022/05/17/data-location-inheritance-bug/
```

(continues on next page)

(continued from previous page)

```
↪",
        "introduced": "0.6.9",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-2",
        "name": "NestedCalldataArrayAbiReencodingSizeValidation",
        "summary": "ABI-reencoding of nested dynamic calldata arrays did not always␣
↪perform proper size checks against the size of calldata and could read beyond␣
↪``calldatasize()``.",
        "description": "Calldata validation for nested dynamic types is deferred until␣
↪the first access to the nested values. Such an access may for example be a copy to␣
↪memory or an index or member access to the outer type. While in most such accesses␣
↪calldata validation correctly checks that the data area of the nested array is␣
↪completely contained in the passed calldata (i.e. in the range [0, calldatasize()]),␣
↪this check may not be performed, when ABI encoding such nested types again directly␣
↪from calldata. For instance, this can happen, if a value in calldata with a nested␣
↪dynamic array is passed to an external call, used in ``abi.encode`` or emitted as␣
↪event. In such cases, if the data area of the nested array extends beyond␣
↪``calldatasize()``, ABI encoding it did not revert, but continued reading values from␣
↪beyond ``calldatasize()`` (i.e. zero values).",
        "link": "https://blog.soliditylang.org/2022/05/17/calldata-reencode-size-check-
↪bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-1",
        "name": "AbiEncodeCallLiteralAsFixedBytesBug",
        "summary": "Literals used for a fixed length bytes parameter in ``abi.
↪encodeCall`` were encoded incorrectly.",
        "description": "For the encoding, the compiler only considered the types of the␣
↪expressions in the second argument of ``abi.encodeCall`` itself, but not the parameter␣
↪types of the function given as first argument. In almost all cases the abi encoding of␣
↪the type of the expression matches the abi encoding of the parameter type of the given␣
↪function. This is because the type checker ensures the expression is implicitly␣
↪convertible to the respective parameter type. However this is not true for number␣
↪literals used for fixed bytes types shorter than 32 bytes, nor for string literals␣
↪used for any fixed bytes type. Number literals were encoded as numbers instead of␣
↪being shifted to become left-aligned. String literals were encoded as dynamically␣
↪sized memory strings instead of being converted to a left-aligned bytes value.",
        "link": "https://blog.soliditylang.org/2022/03/16/encodecall-bug/",
        "introduced": "0.8.11",
        "fixed": "0.8.13",
        "severity": "very low"

    },
    {
        "uid": "SOL-2021-4",
        "name": "UserDefinedValueTypesBug",
```

(continues on next page)

(continued from previous page)

```
        "summary": "User defined value types with underlying type shorter than 32 bytes␣
↪used incorrect storage layout and wasted storage",
        "description": "The compiler did not correctly compute the storage layout of␣
↪user defined value types based on types that are shorter than 32 bytes. It would␣
↪always use a full storage slot for these types, even if the underlying type was␣
↪shorter. This was wasteful and might have problems with tooling or contract upgrades.",
        "link": "https://blog.soliditylang.org/2021/09/29/user-defined-value-types-bug/",
        "introduced": "0.8.8",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-3",
        "name": "SignedImmutables",
        "summary": "Immutable variables of signed integer type shorter than 256 bits can␣
↪lead to values with invalid higher order bits if inline assembly is used.",
        "description": "When immutable variables of signed integer type shorter than 256␣
↪bits are read, their higher order bits were unconditionally set to zero. The correct␣
↪operation would be to sign-extend the value, i.e. set the higher order bits to one if␣
↪the sign bit is one. This sign-extension is performed by Solidity just prior to when␣
↪it matters, i.e. when a value is stored in memory, when it is compared or when a␣
↪division is performed. Because of that, to our knowledge, the only way to access the␣
↪value in its unclean state is by reading it through inline assembly.",
        "link": "https://blog.soliditylang.org/2021/09/29/signed-immutables-bug/",
        "introduced": "0.6.5",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-2",
        "name": "ABIDecodeTwoDimensionalArrayMemory",
        "summary": "If used on memory byte arrays, result of the function ``abi.decode``␣
↪can depend on the contents of memory outside of the actual byte array that is decoded.
↪",
        "description": "The ABI specification uses pointers to data areas for everything␣
↪that is dynamically-sized. When decoding data from memory (instead of calldata), the␣
↪ABI decoder did not properly validate some of these pointers. More specifically, it␣
↪was possible to use large values for the pointers inside arrays such that computing␣
↪the offset resulted in an undetected overflow. This could lead to these pointers␣
↪targeting areas in memory outside of the actual area to be decoded. This way, it was␣
↪possible for ``abi.decode`` to return different values for the same encoded byte array.
↪",
        "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/",
        "introduced": "0.4.16",
        "fixed": "0.8.4",
        "conditions": {
            "ABIEncoderV2": true
        },
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-1",
```

(continues on next page)

(continued from previous page)

```
        "name": "KeccakCaching",
        "summary": "The bytecode optimizer incorrectly reused previously evaluated␣
↪Keccak-256 hashes. You are unlikely to be affected if you do not compute Keccak-256␣
↪hashes in inline assembly.",
        "description": "Solidity's bytecode optimizer has a step that can compute Keccak-
↪256 hashes, if the contents of the memory are known during compilation time. This step␣
↪also has a mechanism to determine that two Keccak-256 hashes are equal even if the␣
↪values in memory are not known during compile time. This mechanism had a bug where␣
↪Keccak-256 of the same memory content, but different sizes were considered equal. More␣
↪specifically, ``keccak256(mpos1, length1)`` and ``keccak256(mpos2, length2)`` in some␣
↪cases were considered equal if ``length1`` and ``length2``, when rounded up to nearest␣
↪multiple of 32 were the same, and when the memory contents at ``mpos1`` and ``mpos2``␣
↪can be deduced to be equal. You maybe affected if you compute multiple Keccak-256␣
↪hashes of the same content, but with different lengths inside inline assembly. You are␣
↪unaffected if your code uses ``keccak256`` with a length that is not a compile-time␣
↪constant or if it is always a multiple of 32.",
        "link": "https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug/",
        "fixed": "0.8.3",
        "conditions": {
            "optimizer": true
        },
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-11",
        "name": "EmptyByteArrayCopy",
        "summary": "Copying an empty byte array (or string) from memory or calldata to␣
↪storage can result in data corruption if the target array's length is increased␣
↪subsequently without storing new data.",
        "description": "The routine that copies byte arrays from memory or calldata to␣
↪storage stores unrelated data from after the source array in the storage slot if the␣
↪source array is empty. If the storage array's length is subsequently increased either␣
↪by using ``.push()`` or by assigning to its ``.length`` attribute (only before 0.6.0),␣
↪the newly created byte array elements will not be zero-initialized, but contain the␣
↪unrelated data. You are not affected if you do not assign to ``.length`` and do not␣
↪use ``.push()`` on byte arrays, or only use ``.push(<arg>)`` or manually initialize␣
↪the new elements.",
        "link": "https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/",
        "fixed": "0.7.4",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-10",
        "name": "DynamicArrayCleanup",
        "summary": "When assigning a dynamically-sized array with types of size at most␣
↪16 bytes in storage causing the assigned array to shrink, some parts of deleted slots␣
↪were not zeroed out.",
        "description": "Consider a dynamically-sized array in storage whose base-type is␣
↪small enough such that multiple values can be packed into a single slot, such as␣
↪`uint128[]`. Let us define its length to be `l`. When this array gets assigned from␣
↪another array with a smaller length, say `m`, the slots between elements `m` and `l`␣
↪have to be cleaned by zeroing them out. However, this cleaning was not performed␣
```

(continues on next page)

(continued from previous page)

```
↪properly. Specifically, after the slot corresponding to `m`, only the first packed␣
↪value was cleaned up. If this array gets resized to a length larger than `m`, the␣
↪indices corresponding to the unclean parts of the slot contained the original value,␣
↪instead of 0. The resizing here is performed by assigning to the array `length`, by a␣
↪`push()` or via inline assembly. You are not affected if you are only using `.push(
↪<arg>)` or if you assign a value (even zero) to the new elements after increasing the␣
↪length of the array.",
        "link": "https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-cleanup-
↪bug/",
        "fixed": "0.7.3",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-9",
        "name": "FreeFunctionRedefinition",
        "summary": "The compiler does not flag an error when two or more free functions␣
↪with the same name and parameter types are defined in a source unit or when an␣
↪imported free function alias shadows another free function with a different name but␣
↪identical parameter types.",
        "description": "In contrast to functions defined inside contracts, free␣
↪functions with identical names and parameter types did not create an error. Both␣
↪definition of free functions with identical name and parameter types and an imported␣
↪free function with an alias that shadows another function with a different name but␣
↪identical parameter types were permitted due to which a call to either the multiply␣
↪defined free function or the imported free function alias within a contract led to the␣
↪execution of that free function which was defined first within the source unit.␣
↪Subsequently defined identical free function definitions were silently ignored and␣
↪their code generation was skipped.",
        "introduced": "0.7.1",
        "fixed": "0.7.2",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-8",
        "name": "UsingForCalldata",
        "summary": "Function calls to internal library functions with calldata␣
↪parameters called via ``using for`` can result in invalid data being read.",
        "description": "Function calls to internal library functions using the ``using␣
↪for`` mechanism copied all calldata parameters to memory first and passed them on like␣
↪that, regardless of whether it was an internal or an external call. Due to that, the␣
↪called function would receive a memory pointer that is interpreted as a calldata␣
↪pointer. Since dynamically sized arrays are passed using two stack slots for calldata,␣
↪but only one for memory, this can lead to stack corruption. An affected library call␣
↪will consider the JUMPDEST to which it is supposed to return as part of its arguments␣
↪and will instead jump out to whatever was on the stack before the call.",
        "introduced": "0.6.9",
        "fixed": "0.6.10",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-7",
        "name": "MissingEscapingInFormatting",
```

(continues on next page)

(continued from previous page)

```
        "summary": "String literals containing double backslash characters passed␣
↪directly to external or encoding function calls can lead to a different string being␣
↪used when ABIEncoderV2 is enabled.",
        "description": "When ABIEncoderV2 is enabled, string literals passed directly to␣
↪encoding functions or external function calls are stored as strings in the␣
↪intermediate code. Characters outside the printable range are handled correctly, but␣
↪backslashes are not escaped in this procedure. This leads to double backslashes being␣
↪reduced to single backslashes and consequently re-interpreted as escapes potentially␣
↪resulting in a different string being encoded.",
        "introduced": "0.5.14",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-6",
        "name": "ArraySliceDynamicallyEncodedBaseType",
        "summary": "Accessing array slices of arrays with dynamically encoded base types␣
↪(e.g. multi-dimensional arrays) can result in invalid data being read.",
        "description": "For arrays with dynamically sized base types, index range␣
↪accesses that use a start expression that is non-zero will result in invalid array␣
↪slices. Any index access to such array slices will result in data being read from␣
↪incorrect calldata offsets. Array slices are only supported for dynamic calldata types␣
↪and all problematic type require ABIEncoderV2 to be enabled.",
        "introduced": "0.6.0",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-5",
        "name": "ImplicitConstructorCallvalueCheck",
        "summary": "The creation code of a contract that does not define a constructor␣
↪but has a base that does define a constructor did not revert for calls with non-zero␣
↪value.",
        "description": "Starting from Solidity 0.4.5 the creation code of contracts␣
↪without explicit payable constructor is supposed to contain a callvalue check that␣
↪results in contract creation reverting, if non-zero value is passed. However, this␣
↪check was missing in case no explicit constructor was defined in a contract at all,␣
↪but the contract has a base that does define a constructor. In these cases it is␣
↪possible to send value in a contract creation transaction or using inline assembly␣
↪without revert, even though the creation code is supposed to be non-payable.",
        "introduced": "0.4.5",
        "fixed": "0.6.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-4",
```

(continues on next page)

(continued from previous page)

```
        "name": "TupleAssignmentMultiStackSlotComponents",
        "summary": "Tuple assignments with components that occupy several stack slots, i.
↪e. nested tuples, pointers to external functions or references to dynamically sized␣
↪calldata arrays, can result in invalid values.",
        "description": "Tuple assignments did not correctly account for tuple components␣
↪that occupy multiple stack slots in case the number of stack slots differs between␣
↪left-hand-side and right-hand-side. This can either happen in the presence of nested␣
↪tuples or if the right-hand-side contains external function pointers or references to␣
↪dynamic calldata arrays, while the left-hand-side contains an omission.",
        "introduced": "0.1.6",
        "fixed": "0.6.6",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-3",
        "name": "MemoryArrayCreationOverflow",
        "summary": "The creation of very large memory arrays can result in overlapping␣
↪memory regions and thus memory corruption.",
        "description": "No runtime overflow checks were performed for the length of␣
↪memory arrays during creation. In cases for which the memory size of an array in bytes,
↪ i.e. the array length times 32, is larger than 2^256-1, the memory allocation will␣
↪overflow, potentially resulting in overlapping memory areas. The length of the array␣
↪is still stored correctly, so copying or iterating over such an array will result in␣
↪out-of-gas.",
        "link": "https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/",
        "introduced": "0.2.0",
        "fixed": "0.6.5",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and continue␣
↪and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared inside␣
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or␣
↪``continue`` even though they cannot be removed in other control-flow branches.␣
↪Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.6.0",
        "fixed": "0.6.1",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2020-2",
        "name": "privateCanBeOverridden",
```

(continues on next page)

(continued from previous page)

```
        "summary": "Private methods can be overridden by inheriting contracts.",
        "description": "While private methods of base contracts are not visible and␣
↪cannot be called directly from the derived contract, it is still possible to declare a␣
↪function of the same name and type and thus change the behaviour of the base contract
↪'s function.",
        "introduced": "0.3.0",
        "fixed": "0.5.17",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and continue␣
↪and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared inside␣
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or␣
↪``continue`` even though they cannot be removed in other control-flow branches.␣
↪Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.5.8",
        "fixed": "0.5.16",
        "severity": "low",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-10",
        "name": "ABIEncoderV2LoopYulOptimizer",
        "summary": "If both the experimental ABIEncoderV2 and the experimental Yul␣
↪optimizer are activated, one component of the Yul optimizer may reuse data in memory␣
↪that has been changed in the meantime.",
        "description": "The Yul optimizer incorrectly replaces ``mload`` and ``sload``␣
↪calls with values that have been previously written to the load location (and␣
↪potentially changed in the meantime) if all of the following conditions are met: (1)␣
↪there is a matching ``mstore`` or ``sstore`` call before; (2) the contents of memory␣
↪or storage is only changed in a function that is called (directly or indirectly) in␣
↪between the first store and the load call; (3) called function contains a for loop␣
↪where the same memory location is changed in the condition or the post or body block.␣
↪When used in Solidity mode, this can only happen if the experimental ABIEncoderV2 is␣
↪activated and the experimental Yul optimizer has been activated manually in addition␣
↪to the regular optimizer in the compiler settings.",
        "introduced": "0.5.14",
        "fixed": "0.5.15",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true,
            "optimizer": true,
            "yulOptimizer": true
```

(continues on next page)

(continued from previous page)

```
        }
    },
    {
        "uid": "SOL-2019-9",
        "name":
↪"ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
        "summary": "Reading from calldata structs that contain dynamically encoded, but␣
↪statically-sized members can result in incorrect values.",
        "description": "When a calldata struct contains a dynamically encoded, but␣
↪statically-sized member, the offsets for all subsequent struct members are calculated␣
↪incorrectly. All reads from such members will result in invalid values. Only calldata␣
↪structs are affected, i.e. this occurs in external functions with such structs as␣
↪argument. Using affected structs in storage or memory or as arguments to public␣
↪functions on the other hand works correctly.",
        "introduced": "0.5.6",
        "fixed": "0.5.11",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-8",
        "name": "SignedArrayStorageCopy",
        "summary": "Assigning an array of signed integers to a storage array of␣
↪different type can lead to data corruption in that array.",
        "description": "In two's complement, negative integers have their higher order␣
↪bits set. In order to fit into a shared storage slot, these have to be set to zero.␣
↪When a conversion is done at the same time, the bits to set to zero were incorrectly␣
↪determined from the source and not the target type. This means that such copy␣
↪operations can lead to incorrect values being stored.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.7",
        "fixed": "0.5.10",
        "severity": "low/medium"
    },
    {
        "uid": "SOL-2019-7",
        "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
        "summary": "Storage arrays containing structs or other statically-sized arrays␣
↪are not read properly when directly encoded in external function calls or in abi.
↪encode*.",
        "description": "When storage arrays whose elements occupy more than a single␣
↪storage slot are directly encoded in external function calls or using abi.encode*,␣
↪their elements are read in an overlapping manner, i.e. the element pointer is not␣
↪properly advanced between reads. This is not a problem when the storage data is first␣
↪copied to a memory variable or if the storage array only contains value types or␣
↪dynamically-sized arrays.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.16",
        "fixed": "0.5.10",
        "severity": "low",
```

(continues on next page)

(continued from previous page)

```
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-6",
        "name": "DynamicConstructorArgumentsClippedABIV2",
        "summary": "A contract's constructor that takes structs or arrays that contain␣
↪dynamically-sized arrays reverts or decodes to invalid data.",
        "description": "During construction of a contract, constructor parameters are␣
↪copied from the code section to memory for decoding. The amount of bytes to copy was␣
↪calculated incorrectly in case all parameters are statically-sized but contain␣
↪dynamically-sized arrays as struct members or inner arrays. Such types are only␣
↪available if ABIEncoderV2 is activated.",
        "introduced": "0.4.16",
        "fixed": "0.5.9",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special jump␣
↪target only considered the situation after deployment.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor_0.4.x",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special jump␣
↪target only considered the situation after deployment.",
        "introduced": "0.4.5",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries",
        "summary": "Contract types used in events in libraries cause an incorrect event␣
↪signature hash",
```

(continues on next page)

(continued from previous page)

```
        "description": "Instead of using the type `address` in the hashed signature, the␣
↪actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries_0.4.x",
        "summary": "Contract types used in events in libraries cause an incorrect event␣
↪signature hash",
        "description": "Instead of using the type `address` in the hashed signature, the␣
↪actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.3.0",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but can␣
↪also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.0",
        "fixed": "0.5.7",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage_0.4.x",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but can␣
↪also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.4.19",
        "fixed": "0.4.26",
        "severity": "low",
        "conditions": {
```

(continues on next page)

(continued from previous page)

```
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-2",
        "name": "IncorrectByteInstructionOptimization",
        "summary": "The optimizer incorrectly handles byte opcodes whose second argument␣
↪is 31 or a constant expression that evaluates to 31. This can result in unexpected␣
↪values.",
        "description": "The optimizer incorrectly handles byte opcodes that use the␣
↪constant 31 as second argument. This can happen when performing index access on␣
↪bytesNN types with a compile-time constant value (not index) of 31 or when using the␣
↪byte opcode in inline assembly.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.7",
        "severity": "very low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2019-1",
        "name": "DoubleShiftSizeOverflow",
        "summary": "Double bitwise shifts by large constants whose sum overflows 256␣
↪bits can result in unexpected values.",
        "description": "Nested logical shift operations whose total shift size is 2**256␣
↪or more are incorrectly optimized. This only applies to shifts by numbers of bits that␣
↪are compile-time constant expressions.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.6",
        "severity": "low",
        "conditions": {
            "optimizer": true,
            "evmVersion": ">=constantinople"
        }
    },
    {
        "uid": "SOL-2018-4",
        "name": "ExpExponentCleanup",
        "summary": "Using the ** operator with an exponent of type shorter than 256 bits␣
↪can result in unexpected values.",
        "description": "Higher order bits in the exponent are not properly cleaned␣
↪before the EXP opcode is applied if the type of the exponent expression is smaller␣
↪than 256 bits and not smaller than the type of the base. In that case, the result␣
↪might be larger than expected if the exponent is assumed to lie within the value range␣
↪of the type. Literal numbers as exponents are unaffected as are exponents or bases of␣
↪type uint256.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
```

(continues on next page)

(continued from previous page)

```
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "uid": "SOL-2018-3",
        "name": "EventStructWrongData",
        "summary": "Using structs in events logged wrong data.",
        "description": "If a struct is used in an event, the address of the struct is␣
↪logged instead of the actual data.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..[?
↪(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↪'struct'))]"}
    },
    {
        "uid": "SOL-2018-2",
        "name": "NestedArrayFunctionCallDecoder",
        "summary": "Calling functions that return multi-dimensional fixed-size arrays␣
↪can result in memory corruption.",
        "description": "If Solidity code calls a function that returns a multi-
↪dimensional fixed-size array, array elements are incorrectly interpreted as memory␣
↪pointers and thus can cause memory corruption if the return values are accessed.␣
↪Calling functions with multi-dimensional fixed-size arrays is unaffected as is␣
↪returning fixed-size arrays from function calls. The regular expression only checks if␣
↪such functions are present, not if they are called, which is required for the contract␣
↪to be affected.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.1.4",
        "fixed": "0.4.22",
        "severity": "medium",
        "check": {"regex-source": "returns[^;{]*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\]\\
↪s*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "uid": "SOL-2018-1",
        "name": "OneOfTwoConstructorsSkipped",
        "summary": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored.",
        "description": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored. There will be a compiler␣
↪warning about the old-style constructor, so contracts only using new-style␣
↪constructors are fine.",
        "introduced": "0.4.22",
        "fixed": "0.4.23",
        "severity": "very low"
    },
```

(continues on next page)

(continued from previous page)

```
    {
        "uid": "SOL-2017-5",
        "name": "ZeroFunctionSelector",
        "summary": "It is possible to craft the name of a function such that it is␣
↪executed instead of the fallback function in very specific circumstances.",
        "description": "If a function has a selector consisting only of zeros, is␣
↪payable and part of a contract that does not have a fallback function and at most five␣
↪external functions in total, this function is called instead of the fallback function␣
↪if Ether is sent to the contract without data.",
        "fixed": "0.4.18",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-4",
        "name": "DelegateCallReturnValue",
        "summary": "The low-level .delegatecall() does not return the execution outcome,␣
↪but converts the value returned by the functioned called to a boolean instead.",
        "description": "The return value of the low-level .delegatecall() function is␣
↪taken from a position in memory, where the call data or the return data resides. This␣
↪value is interpreted as a boolean and put onto the stack. This means if the called␣
↪function returns at least 32 zero bytes, .delegatecall() returns false even if the␣
↪call was successful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-3",
        "name": "ECRecoverMalformedInput",
        "summary": "The ecrecover() builtin can return garbage for malformed input.",
        "description": "The ecrecover precompile does not properly signal failure for␣
↪malformed input (especially in the 'v' argument) and thus the Solidity function can␣
↪return data that was previously present in the return area in memory.",
        "fixed": "0.4.14",
        "severity": "medium"
    },
    {
        "uid": "SOL-2017-2",
        "name": "SkipEmptyStringLiteral",
        "summary": "If \"\" is used in a function call, the following function arguments␣
↪will not be correctly passed to the function.",
        "description": "If the empty string literal \"\" is used as an argument in a␣
↪function call, it is skipped by the encoder. This has the effect that the encoding of␣
↪all arguments following this is shifted left by 32 bytes and thus the function call␣
↪data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-1",
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
```

(continues on next page)

(continued from previous page)

```
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode by␣
↪routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.soliditylang.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-11",
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying memory,
↪ ignored its return value. On the public chain, calls to the identity precompile can␣
↪be made in a way that they never fail, but this might be different on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "uid": "SOL-2016-10",
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At␣
↪jump destinations, multiple code paths join and thus it has to compute a common state␣
↪from the incoming edges. Computing this common state was simplified to just use the␣
↪empty state, but this implementation was not done properly. This bug can cause data␣
↪corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-9",
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly and␣
↪could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same 32␣
↪byte storage slot, but storage writes always write 32 bytes. For some types, the␣
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
        "link": "https://blog.soliditylang.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
```

(continues on next page)

(continued from previous page)

```
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "uid": "SOL-2016-8",
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not properly␣
↪reset across control flow paths and thus the optimizer sometimes thought that the␣
↪result of a SHA3 operation is already present on the stack. This could result in data␣
↪corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-7",
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how much␣
↪Ether was sent with a call, the library function incorrectly assumed that Ether was␣
↪sent to the library and threw an exception.",
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "uid": "SOL-2016-6",
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if no␣
↪Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "uid": "SOL-2016-5",
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite loop␣
↪and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this length␣
↪is zero, code was generated that did not terminate and thus consumed all gas.",
        "severity": "low",
```

(continues on next page)

(continued from previous page)

```
        "fixed": "0.3.6"
    },
    {
        "uid": "SOL-2016-4",
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At␣
↪jump destinations, multiple code paths join and thus it has to compute a common state␣
↪from the incoming edges. Computing this common state was not done correctly. This bug␣
↪can cause data corruption, but it is probably quite hard to use for targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-3",
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned before␣
↪comparison.",
        "description": "Two variables of type bytesNN were considered different if their␣
↪higher order bits, which are not part of the actual value, were different. An attacker␣
↪might use this to reach seemingly unreachable code paths by providing incorrectly␣
↪formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "uid": "SOL-2016-2",
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32 bytes␣
↪did not correctly clean the higher order bits, causing corruption in other array␣
↪elements.",
        "description": "Multiple elements of an array of values that are shorter than 17␣
↪bytes are packed into the same storage slot. Writing to a single element of such an␣
↪array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "uid": "SOL-2016-1",
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
```

(continues on next page)

(continued from previous page)

```
]
```

3.26 Solidity v0.5.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.5.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

Note

Contracts compiled with Solidity v0.5.0 can still interface with contracts and even libraries compiled with older versions without recompiling or redeploying them. Changing the interfaces to include data locations and visibility and mutability specifiers suffices. See the *Interoperability With Older Contracts* section below.

3.26.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- Signed right shift now uses proper arithmetic shift, i.e. rounding towards negative infinity, instead of rounding towards zero. Signed and unsigned shift will have dedicated opcodes in Constantinople, and are emulated by Solidity for the moment.
- The `continue` statement in a `do...while` loop now jumps to the condition, which is the common behavior in such cases. It used to jump to the loop body. Thus, if the condition is false, the loop terminates.
- The functions `.call()`, `.delegatecall()` and `.staticcall()` do not pad anymore when given a single `bytes` parameter.
- Pure and view functions are now called using the opcode `STATICCALL` instead of `CALL` if the EVM version is Byzantium or later. This disallows state changes on the EVM level.
- The ABI encoder now properly pads byte arrays and strings from calldata (`msg.data` and external function parameters) when used in external function calls and in `abi.encode`. For unpadded encoding, use `abi.encodePacked`.
- The ABI decoder reverts in the beginning of functions and in `abi.decode()` if passed calldata is too short or points out of bounds. Note that dirty higher order bits are still simply ignored.
- Forward all available gas with external function calls starting from Tangerine Whistle.

3.26.2 Semantic and Syntactic Changes

This section highlights changes that affect syntax and semantics.

- The functions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` and `ripemd160()` now accept only a single `bytes` argument. Moreover, the argument is not padded. This was changed to make more explicit and clear how the arguments are concatenated. Change every `.call()` (and family) to a `.call("")` and every `.call(signature, a, b, c)` to use `.call(abi.encodeWithSignature(signature, a, b, c))` (the last one only works for value types). Change every `keccak256(a, b, c)` to `keccak256(abi.encodePacked(a, b, c))`. Even though it is not a breaking change, it is suggested that developers change `x.call(bytes4(keccak256("f(uint256)")), a, b)` to `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
- Functions `.call()`, `.delegatecall()` and `.staticcall()` now return `(bool, bytes memory)` to provide access to the return data. Change `bool success = otherContract.call("f")` to `(bool success, bytes memory data) = otherContract.call("f")`.

- Solidity now implements C99-style scoping rules for function local variables, that is, variables can only be used after they have been declared and only in the same or nested scopes. Variables declared in the initialization block of a `for` loop are valid at any point inside the loop.

3.26.3 Explicitness Requirements

This section lists changes where the code now needs to be more explicit. For most of the topics the compiler will provide suggestions.

- Explicit function visibility is now mandatory. Add `public` to every function and constructor, and `external` to every fallback or interface function that does not specify its visibility already.
- Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables. For example, change `uint[] x = z` to `uint[] storage x = z`, and `function f(uint[][] x)` to `function f(uint[][] memory x)` where `memory` is the data location and might be replaced by `storage` or `calldata` accordingly. Note that `external` functions require parameters with a data location of `calldata`.
- Contract types do not include `address` members anymore in order to separate the namespaces. Therefore, it is now necessary to explicitly convert values of contract type to addresses before using an `address` member. Example: if `c` is a contract, change `c.transfer(...)` to `address(c).transfer(...)`, and `c.balance` to `address(c).balance`.
- Explicit conversions between unrelated contract types are now disallowed. You can only convert from a contract type to one of its base or ancestor types. If you are sure that a contract is compatible with the contract type you want to convert to, although it does not inherit from it, you can work around this by converting to `address` first. Example: if `A` and `B` are contract types, `B` does not inherit from `A` and `b` is a contract of type `B`, you can still convert `b` to type `A` using `A(address(b))`. Note that you still need to watch out for matching payable fallback functions, as explained below.
- The `address` type was split into `address` and `address payable`, where only `address payable` provides the `transfer` function. An `address payable` can be directly converted to an `address`, but the other way around is not allowed. Converting `address` to `address payable` is possible via conversion through `uint160`. If `c` is a contract, `address(c)` results in `address payable` only if `c` has a payable fallback function. If you use the *withdraw pattern*, you most likely do not have to change your code because `transfer` is only used on `msg.sender` instead of stored addresses and `msg.sender` is an `address payable`.
- Conversions between `bytesX` and `uintY` of different size are now disallowed due to `bytesX` padding on the right and `uintY` padding on the left which may cause unexpected conversion results. The size must now be adjusted within the type before the conversion. For example, you can convert a `bytes4` (4 bytes) to a `uint64` (8 bytes) by first converting the `bytes4` variable to `bytes8` and then to `uint64`. You get the opposite padding when converting through `uint32`. Before v0.5.0 any conversion between `bytesX` and `uintY` would go through `uint8X`. For example `uint8(bytes3(0x291807))` would be converted to `uint8(uint24(bytes3(0x291807)))` (the result is `0x07`).
- Using `msg.value` in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into `payable` or create a new internal function for the program logic that uses `msg.value`.
- For clarity reasons, the command-line interface now requires `-` if the standard input is used as source.

3.26.4 Deprecated Elements

This section lists changes that deprecate prior features or syntax. Note that many of these changes were already enabled in the experimental mode `v0.5.0`.

Command-line and JSON Interfaces

- The command-line option `--formal` (used to generate Why3 output for further formal verification) was deprecated and is now removed. A new formal verification module, the SMTChecker, is enabled via `pragma experimental SMTChecker;`.
- The command-line option `--julia` was renamed to `--yul` due to the renaming of the intermediate language `Julia` to `Yul`.
- The `--clone-bin` and `--combined-json clone-bin` command-line options were removed.
- Remappings with empty prefix are disallowed.
- The JSON AST fields `constant` and `payable` were removed. The information is now present in the `stateMutability` field.
- The JSON AST field `isConstructor` of the `FunctionDefinition` node was replaced by a field called `kind` which can have the value `"constructor"`, `"fallback"` or `"function"`.
- In unlinked binary hex files, library address placeholders are now the first 36 hex characters of the keccak256 hash of the fully qualified library name, surrounded by `$...$`. Previously, just the fully qualified library name was used. This reduces the chances of collisions, especially when long paths are used. Binary files now also contain a list of mappings from these placeholders to the fully qualified names.

Constructors

- Constructors must now be defined using the `constructor` keyword.
- Calling base constructors without parentheses is now disallowed.
- Specifying base constructor arguments multiple times in the same inheritance hierarchy is now disallowed.
- Calling a constructor with arguments but with wrong argument count is now disallowed. If you only want to specify an inheritance relation without giving arguments, do not provide parentheses at all.

Functions

- Function `callcode` is now disallowed (in favor of `delegatecall`). It is still possible to use it via inline assembly.
- `suicide` is now disallowed (in favor of `selfdestruct`).
- `sha3` is now disallowed (in favor of `keccak256`).
- `throw` is now disallowed (in favor of `revert`, `require` and `assert`).

Conversions

- Explicit and implicit conversions from decimal literals to `bytesXX` types is now disallowed.
- Explicit and implicit conversions from hex literals to `bytesXX` types of different size is now disallowed.

Literals and Suffixes

- The unit denomination `years` is now disallowed due to complications and confusions about leap years.
- Trailing dots that are not followed by a number are now disallowed.
- Combining hex numbers with unit denominations (e.g. `0x1e wei`) is now disallowed.
- The prefix `0X` for hex numbers is disallowed, only `0x` is possible.

Variables

- Declaring empty structs is now disallowed for clarity.
- The `var` keyword is now disallowed to favor explicitness.
- Assignments between tuples with different number of components is now disallowed.
- Values for constants that are not compile-time constants are disallowed.
- Multi-variable declarations with mismatching number of values are now disallowed.
- Uninitialized storage variables are now disallowed.
- Empty tuple components are now disallowed.
- Detecting cyclic dependencies in variables and structs is limited in recursion to 256.
- Fixed-size arrays with a length of zero are now disallowed.

Syntax

- Using `constant` as function state mutability modifier is now disallowed.
- Boolean expressions cannot use arithmetic operations.
- The unary + operator is now disallowed.
- Literals cannot anymore be used with `abi.encodePacked` without prior conversion to an explicit type.
- Empty return statements for functions with one or more return values are now disallowed.
- The "loose assembly" syntax is now disallowed entirely, that is, jump labels, jumps and non-functional instructions cannot be used anymore. Use the new `while`, `switch` and `if` constructs instead.
- Functions without implementation cannot use modifiers anymore.
- Function types with named return values are now disallowed.
- Single statement variable declarations inside if/while/for bodies that are not blocks are now disallowed.
- New keywords: `calldata` and `constructor`.
- New reserved keywords: `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` and `unchecked`.

3.26.5 Interoperability With Older Contracts

It is still possible to interface with contracts written for Solidity versions prior to v0.5.0 (or the other way around) by defining interfaces for them. Consider you have the following pre-0.5.0 contract already deployed:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will report a warning until version 0.4.25 of the compiler
// This will not compile after 0.5.0
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

This will no longer compile with Solidity v0.5.0. However, you can define a compatible interface for it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Note that we did not declare `anotherOldFunction` to be `view`, despite it being declared `constant` in the original contract. This is due to the fact that starting with Solidity v0.5.0 `staticcall` is used to call `view` functions. Prior to v0.5.0 the `constant` keyword was not enforced, so calling a function declared `constant` with `staticcall` may still revert, since the `constant` function may still attempt to modify storage. Consequently, when defining an interface for older contracts, you should only use `view` in place of `constant` in case you are absolutely sure that the function will work with `staticcall`.

Given the interface defined above, you can now easily use the already deployed pre-0.5.0 contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

Similarly, pre-0.5.0 libraries can be used by defining the functions of the library without implementation and supplying the address of the pre-0.5.0 library during linking (see *Using the Commandline Compiler* for how to use the commandline compiler for linking):

```
// This will not compile after 0.6.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

3.26.6 Example

The following example shows a contract and its updated version for Solidity v0.5.0 with some of the changes listed in this section.

Old version:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will not compile after 0.5.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Function mutability not provided, not an error.
    function someInteger() internal returns (uint) { return 2; }

    // Function visibility not provided, not an error.
    // Function mutability not provided, not an error.
    function f(uint x) returns (bytes) {
        // Var is fine in this version.
        var z = someInteger();
        x += z;
        // Throw is fine in this version.
        if (x > 100)
            throw;
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (wrong, should be -2)
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' causes an infinite loop.
        } while (x < 11);
        // Call returns only a Bool.
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // Local variables could be declared after their use.
            int y;
        }
        return b;
    }
```

(continues on next page)

(continued from previous page)

```
    // No need for an explicit data location for 'arr'
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the first 4 bytes of x will be lost. This might lead to
        // unexpected behavior since bytesX are right padded.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

New version:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;
// This will not compile after 0.6.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' jumps to the condition below.
        } while (x < 11);

        // Call returns (bool, bytes).
        // Data location must be specified.
        (bool success, bytes memory data) = address(other).call("f");
```

(continues on next page)

(continued from previous page)

```
        if (!success)
            revert();
        return data;
    }

    using AddressMakePayable for address;
    // Data location for 'arr' must be specified
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract, address␣
↪unknownContract) public payable {
        // 'otherContract.transfer' is not provided.
        // Since the code of 'OtherContract' is known and has the fallback
        // function, address(otherContract) has type 'address payable'.
        address(otherContract).transfer(1 ether);

        // 'unknownContract.transfer' is not provided.
        // 'address(unknownContract).transfer' is not provided
        // since 'address(unknownContract)' is not 'address payable'.
        // If the function takes an 'address' which you want to send
        // funds to, you can convert it to 'address payable' via 'uint160'.
        // Note: This is not recommended and the explicit type
        // 'address payable' should be used whenever possible.
        // To increase clarity, we suggest the use of a library for
        // the conversion (provided after the contract in this example).
        address payable addr = unknownContract.makePayable();
        require(addr.send(1 ether));

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the conversion is not allowed.
        // We need to convert to a common size first:
        bytes4 x4 = bytes4(x); // Padding happens on the right
        uint32 y = uint32(x4); // Conversion is consistent
        // 'msg.value' cannot be used in a 'non-payable' function.
        // We need to make the function payable
        myNumber += y + msg.value;
    }
}

// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library AddressMakePayable {
    function makePayable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

3.27 Solidity v0.6.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.6.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

3.27.1 Changes the Compiler Might not Warn About

This section lists changes where the behavior of your code might change without the compiler telling you about it.

- The resulting type of an exponentiation is the type of the base. It used to be the smallest type that can hold both the type of the base and the type of the exponent, as with symmetric operations. Additionally, signed types are allowed for the base of the exponentiation.

3.27.2 Explicitness Requirements

This section lists changes where the code now needs to be more explicit, but the semantics do not change. For most of the topics the compiler will provide suggestions.

- Functions can now only be overridden when they are either marked with the `virtual` keyword or defined in an interface. Functions without implementation outside an interface have to be marked `virtual`. When overriding a function or modifier, the new keyword `override` must be used. When overriding a function or modifier defined in multiple parallel bases, all bases must be listed in parentheses after the keyword like so: `override(Base1, Base2)`.
- Member-access to `length` of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays by assigning a new value to their length. Use `push()`, `push(value)` or `pop()` instead, or assign a full array, which will of course overwrite the existing content. The reason behind this is to prevent storage collisions of gigantic storage arrays.
- The new keyword `abstract` can be used to mark contracts as abstract. It has to be used if a contract does not implement all its functions. Abstract contracts cannot be created using the `new` operator, and it is not possible to generate bytecode for them during compilation.
- Libraries have to implement all their functions, not only the internal ones.
- The names of variables declared in inline assembly may no longer end in `_slot` or `_offset`.
- Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.
- In inline assembly, opcodes that do not take arguments are now represented as "built-in functions" instead of standalone identifiers. So `gas` is now `gas()`.
- State variable shadowing is now disallowed. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

3.27.3 Semantic and Syntactic Changes

This section lists changes where you have to modify your code and it does something else afterwards.

- Conversions from external function types to `address` are now disallowed. Instead external function types have a member called `address`, similar to the existing `selector` member.
- The function `push(value)` for dynamic storage arrays does not return the new length anymore (it returns nothing).
- The unnamed function commonly referred to as "fallback function" was split up into a new fallback function that is defined using the `fallback` keyword and a receive ether function defined using the `receive` keyword.
  - If present, the receive ether function is called whenever the call data is empty (whether or not ether is received). This function is implicitly `payable`.
  - The new fallback function is called when no other function matches (if the receive ether function does not exist then this includes calls with empty call data). You can make this function `payable` or not. If it is not `payable` then transactions not matching any other function which send value will revert. You should only need to implement the new fallback function if you are following an upgrade or proxy pattern.

3.27.4 New Features

This section lists things that were not possible prior to Solidity 0.6.0 or were more difficult to achieve.

- The *try/catch statement* allows you to react on failed external calls.
- `struct` and `enum` types can be declared at file level.
- Array slices can be used for calldata arrays, for example `abi.decode(msg.data[4:], (uint, uint))` is a low-level way to decode the function call payload.
- Natspec supports multiple return parameters in developer documentation, enforcing the same naming check as `@param`.
- Yul and Inline Assembly have a new statement called `leave` that exits the current function.
- Conversions from `address` to `address payable` are now possible via `payable(x)`, where `x` must be of type `address`.

3.27.5 Interface Changes

This section lists changes that are unrelated to the language itself, but that have an effect on the interfaces of the compiler. These may change the way how you use the compiler on the command-line, how you use its programmable interface, or how you analyze the output produced by it.

New Error Reporter

A new error reporter was introduced, which aims at producing more accessible error messages on the command-line. It is enabled by default, but passing `--old-reporter` falls back to the deprecated old error reporter.

Metadata Hash Options

The compiler now appends the IPFS hash of the metadata file to the end of the bytecode by default (for details, see documentation on *contract metadata*). Before 0.6.0, the compiler appended the Swarm hash by default, and in order to still support this behavior, the new command-line option `--metadata-hash` was introduced. It allows you to select the hash to be produced and appended, by passing either `ipfs` or `swarm` as value to the `--metadata-hash` command-line option. Passing the value `none` completely removes the hash.

These changes can also be used via the *Standard JSON Interface* and effect the metadata JSON generated by the compiler.

The recommended way to read the metadata is to read the last two bytes to determine the length of the CBOR encoding and perform a proper decoding on that data block as explained in the *metadata section*.

Yul Optimizer

Together with the legacy bytecode optimizer, the *Yul* optimizer is now enabled by default when you call the compiler with `--optimize`. It can be disabled by calling the compiler with `--no-optimize-yul`. This mostly affects code that uses ABI coder v2.

C API Changes

The client code that uses the C API of `libsolc` is now in control of the memory used by the compiler. To make this change consistent, `solidity_free` was renamed to `solidity_reset`, the functions `solidity_alloc` and `solidity_free` were added and `solidity_compile` now returns a string that must be explicitly freed via `solidity_free()`.

3.27.6 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

- Change `address(f)` to `f.address` for `f` being of external function type.
- Replace `function () external [payable] { ... }` by either `receive() external payable { ... }`, `fallback() external [payable] { ... }` or both. Prefer using a `receive` function only, whenever possible.
- Change `uint length = array.push(value)` to `array.push(value);`. The new length can be accessed via `array.length`.
- Change `array.length++` to `array.push()` to increase, and use `pop()` to decrease the length of a storage array.
- For every named return parameter in a function's `@dev` documentation define a `@return` entry which contains the parameter's name as the first word. E.g. if you have function `f()` defined like `function f() public returns (uint value)` and a `@dev` annotating it, document its return parameters like so: `@return value The return value.`. You can mix named and un-named return parameters documentation so long as the notices are in the order they appear in the tuple return type.
- Choose unique identifiers for variable declarations in inline assembly that do not conflict with declarations outside the inline assembly block.
- Add `virtual` to every non-interface function you intend to override. Add `virtual` to all functions without implementation outside interfaces. For single inheritance, add `override` to every overriding function. For multiple inheritance, add `override(A, B, ..)`, where you list all contracts that define the overridden function in the parentheses. When multiple bases define the same function, the inheriting contract must override all conflicting functions.
- In inline assembly, add `()` to all opcodes that do not otherwise accept an argument. For example, change `pc` to `pc()`, and `gas` to `gas()`.

3.28 Solidity v0.7.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.7.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

3.28.1 Silent Changes of the Semantics

- Exponentiation and shifts of literals by non-literals (e.g. `1 << x` or `2 ** x`) will always use either the type `uint256` (for non-negative literals) or `int256` (for negative literals) to perform the operation. Previously, the operation was performed in the type of the shift amount / the exponent which can be misleading.

3.28.2 Changes to the Syntax

- In external function and contract creation calls, Ether and gas is now specified using a new syntax: `x.f{gas: 10000, value: 2 ether}(arg1, arg2)`. The old syntax – `x.f.gas(10000).value(2 ether)(arg1, arg2)` – will cause an error.
- The global variable `now` is deprecated, `block.timestamp` should be used instead. The single identifier `now` is too generic for a global variable and could give the impression that it changes during transaction processing, whereas `block.timestamp` correctly reflects the fact that it is just a property of the block.
- NatSpec comments on variables are only allowed for public state variables and not for local or internal variables.
- The token `gwei` is a keyword now (used to specify, e.g. `2 gwei` as a number) and cannot be used as an identifier.
- String literals now can only contain printable ASCII characters and this also includes a variety of escape sequences, such as hexadecimal (`\xff`) and unicode escapes (`\u20ac`).

- Unicode string literals are supported now to accommodate valid UTF-8 sequences. They are identified with the `unicode` prefix: `unicode"Hello "`.
- State Mutability: The state mutability of functions can now be restricted during inheritance: Functions with default state mutability can be overridden by `pure` and `view` functions while `view` functions can be overridden by `pure` functions. At the same time, public state variables are considered `view` and even `pure` if they are constants.

Inline Assembly

- Disallow `.` in user-defined function and variable names in inline assembly. It is still valid if you use Solidity in Yul-only mode.
- Slot and offset of storage pointer variable `x` are accessed via `x.slot` and `x.offset` instead of `x_slot` and `x_offset`.

3.28.3 Removal of Unused or Unsafe Features

Mappings outside Storage

- If a struct or array contains a mapping, it can only be used in storage. Previously, mapping members were silently skipped in memory, which is confusing and error-prone.
- Assignments to structs or arrays in storage does not work if they contain mappings. Previously, mappings were silently skipped during the copy operation, which is misleading and error-prone.

Functions and Events

- Visibility (`public` / `internal`) is not needed for constructors anymore: To prevent a contract from being created, it can be marked `abstract`. This makes the visibility concept for constructors obsolete.
- Type Checker: Disallow `virtual` for library functions: Since libraries cannot be inherited from, library functions should not be virtual.
- Multiple events with the same name and parameter types in the same inheritance hierarchy are disallowed.
- `using A for B` only affects the contract it is mentioned in. Previously, the effect was inherited. Now, you have to repeat the `using` statement in all derived contracts that make use of the feature.

Expressions

- Shifts by signed types are disallowed. Previously, shifts by negative amounts were allowed, but reverted at runtime.
- The `finney` and `szabo` denominations are removed. They are rarely used and do not make the actual amount readily visible. Instead, explicit values like `1e20` or the very common `gwei` can be used.

Declarations

- The keyword `var` cannot be used anymore. Previously, this keyword would parse but result in a type error and a suggestion about which type to use. Now, it results in a parser error.

3.28.4 Interface Changes

- JSON AST: Mark hex string literals with `kind: "hexString"`.
- JSON AST: Members with value `null` are removed from JSON output.
- NatSpec: Constructors and functions have consistent userdoc output.

3.28.5 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

- Change `x.f.value(...)()` to `x.f{value: ...}()`. Similarly `(new C).value(...)()` to `new C{value: ...}()` and `x.f.gas(...).value(...)()` to `x.f{gas: ..., value: ...}()`.
- Change `now` to `block.timestamp`.
- Change types of right operand in shift operators to unsigned types. For example change `x >> (256 - y)` to `x >> uint(256 - y)`.
- Repeat the `using A for B` statements in all derived contracts if needed.
- Remove the `public` keyword from every constructor.
- Remove the `internal` keyword from every constructor and add `abstract` to the contract (if not already present).
- Change `_slot` and `_offset` suffixes in inline assembly to `.slot` and `.offset`, respectively.

3.29 Solidity v0.8.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.8.0. For the full list check the release changelog.

3.29.1 Silent Changes of the Semantics

This section lists changes where existing code changes its behavior without the compiler notifying you about it.

- Arithmetic operations revert on underflow and overflow. You can use `unchecked { ... }` to use the previous wrapping behavior.

  Checks for overflow are very common, so we made them the default to increase readability of code, even if it comes at a slight increase of gas costs.

- ABI coder v2 is activated by default.

  You can choose to use the old behavior using `pragma abicoder v1;`. The pragma `pragma experimental ABIEncoderV2;` is still valid, but it is deprecated and has no effect. If you want to be explicit, please use `pragma abicoder v2;` instead.

  Note that ABI coder v2 supports more types than v1 and performs more sanity checks on the inputs. ABI coder v2 makes some function calls more expensive and it can also make contract calls revert that did not revert with ABI coder v1 when they contain data that does not conform to the parameter types.

- Exponentiation is right associative, i.e., the expression `a**b**c` is parsed as `a**(b**c)`. Before 0.8.0, it was parsed as `(a**b)**c`.

  This is the common way to parse the exponentiation operator.

- Failing assertions and other internal checks like division by zero or arithmetic overflow do not use the invalid opcode but instead the revert opcode. More specifically, they will use error data equal to a function call to `Panic(uint256)` with an error code specific to the circumstances.

  This will save gas on errors while it still allows static analysis tools to distinguish these situations from a revert on invalid input, like a failing `require`.

- If a byte array in storage is accessed whose length is encoded incorrectly, a panic is caused. A contract cannot get into this situation unless inline assembly is used to modify the raw representation of storage byte arrays.

- If constants are used in array length expressions, previous versions of Solidity would use arbitrary precision in all branches of the evaluation tree. Now, if constant variables are used as intermediate expressions, their values will be properly rounded in the same way as when they are used in run-time expressions.

- The type `byte` has been removed. It was an alias of `bytes1`.

3.29.2 New Restrictions

This section lists changes that might cause existing contracts to not compile anymore.

- There are new restrictions related to explicit conversions of literals. The previous behavior in the following cases was likely ambiguous:
  1. Explicit conversions from negative literals and literals larger than `type(uint160).max` to `address` are disallowed.
  2. Explicit conversions between literals and an integer type `T` are only allowed if the literal lies between `type(T).min` and `type(T).max`. In particular, replace usages of `uint(-1)` with `type(uint).max`.
  3. Explicit conversions between literals and enums are only allowed if the literal can represent a value in the enum.
  4. Explicit conversions between literals and `address` type (e.g. `address(literal)`) have the type `address` instead of `address payable`. One can get a payable address type by using an explicit conversion, i.e., `payable(literal)`.
- *Address literals* have the type `address` instead of `address payable`. They can be converted to `address payable` by using an explicit conversion, e.g. `payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`.
- There are new restrictions on explicit type conversions. The conversion is only allowed when there is at most one change in sign, width or type-category (`int`, `address`, `bytesNN`, etc.). To perform multiple changes, use multiple conversions.

  Let us use the notation `T(S)` to denote the explicit conversion `T(x)`, where, `T` and `S` are types, and `x` is any arbitrary variable of type `S`. An example of such a disallowed conversion would be `uint16(int8)` since it changes both width (8 bits to 16 bits) and sign (signed integer to unsigned integer). In order to do the conversion, one has to go through an intermediate type. In the previous example, this would be `uint16(uint8(int8))` or `uint16(int16(int8))`. Note that the two ways to convert will produce different results e.g., for `-1`. The following are some examples of conversions that are disallowed by this rule.

  - `address(uint)` and `uint(address)`: converting both type-category and width. Replace this by `address(uint160(uint))` and `uint(uint160(address))` respectively.
  - `payable(uint160)`, `payable(bytes20)` and `payable(integer-literal)`: converting both type-category and state-mutability. Replace this by `payable(address(uint160))`, `payable(address(bytes20))` and `payable(address(integer-literal))` respectively. Note that `payable(0)` is valid and is an exception to the rule.
  - `int80(bytes10)` and `bytes10(int80)`: converting both type-category and sign. Replace this by `int80(uint80(bytes10))` and `bytes10(uint80(int80)` respectively.
  - `Contract(uint)`: converting both type-category and width. Replace this by `Contract(address(uint160(uint)))`.

  These conversions were disallowed to avoid ambiguity. For example, in the expression `uint16 x = uint16(int8(-1))`, the value of `x` would depend on whether the sign or the width conversion was applied first.
- Function call options can only be given once, i.e. `c.f{gas: 10000}{value: 1}()` is invalid and has to be changed to `c.f{gas: 10000, value: 1}()`.
- The global functions `log0`, `log1`, `log2`, `log3` and `log4` have been removed.

  These are low-level functions that were largely unused. Their behavior can be accessed from inline assembly.

- `enum` definitions cannot contain more than 256 members.

  This will make it safe to assume that the underlying type in the ABI is always `uint8`.

- Declarations with the name `this`, `super` and `_` are disallowed, with the exception of public functions and events. The exception is to make it possible to declare interfaces of contracts implemented in languages other than Solidity that do permit such function names.

- Remove support for the `\b`, `\f`, and `\v` escape sequences in code. They can still be inserted via hexadecimal escapes, e.g. `\x08`, `\x0c`, and `\x0b`, respectively.

- The global variables `tx.origin` and `msg.sender` have the type `address` instead of `address payable`. One can convert them into `address payable` by using an explicit conversion, i.e., `payable(tx.origin)` or `payable(msg.sender)`.

  This change was done since the compiler cannot determine whether or not these addresses are payable or not, so it now requires an explicit conversion to make this requirement visible.

- Explicit conversion into `address` type always returns a non-payable `address` type. In particular, the following explicit conversions have the type `address` instead of `address payable`:

  - `address(u)` where `u` is a variable of type `uint160`. One can convert `u` into the type `address payable` by using two explicit conversions, i.e., `payable(address(u))`.
  - `address(b)` where `b` is a variable of type `bytes20`. One can convert `b` into the type `address payable` by using two explicit conversions, i.e., `payable(address(b))`.
  - `address(c)` where `c` is a contract. Previously, the return type of this conversion depended on whether the contract can receive Ether (either by having a receive function or a payable fallback function). The conversion `payable(c)` has the type `address payable` and is only allowed when the contract `c` can receive Ether. In general, one can always convert `c` into the type `address payable` by using the following explicit conversion: `payable(address(c))`. Note that `address(this)` falls under the same category as `address(c)` and the same rules apply for it.

- The `chainid` builtin in inline assembly is now considered `view` instead of `pure`.

- Unary negation cannot be used on unsigned integers anymore, only on signed integers.

3.29.3 Interface Changes

- The output of `--combined-json` has changed: JSON fields `abi`, `devdoc`, `userdoc` and `storage-layout` are sub-objects now. Before 0.8.0 they used to be serialised as strings.

- The "legacy AST" has been removed (`--ast-json` on the commandline interface and `legacyAST` for standard JSON). Use the "compact AST" (`--ast-compact-json` resp. `AST`) as replacement.

- The old error reporter (`--old-reporter`) has been removed.

3.29.4 How to update your code

- If you rely on wrapping arithmetic, surround each operation with `unchecked { ... }`.

- Optional: If you use SafeMath or a similar library, change `x.add(y)` to `x + y`, `x.mul(y)` to `x * y` etc.

- Add `pragma abicoder v1;` if you want to stay with the old ABI coder.

- Optionally remove `pragma experimental ABIEncoderV2` or `pragma abicoder v2` since it is redundant.

- Change `byte` to `bytes1`.

- Add intermediate explicit type conversions if required.

- Combine `c.f{gas: 10000}{value: 1}()` to `c.f{gas: 10000, value: 1}()`.

- Change `msg.sender.transfer(x)` to `payable(msg.sender).transfer(x)` or use a stored variable of `address payable` type.
- Change `x**y**z` to `(x**y)**z`.
- Use inline assembly as a replacement for `log0`, ..., `log4`.
- Negate unsigned integers by subtracting them from the maximum value of the type and adding 1 (e.g. `type(uint256).max - x + 1`, while ensuring that `x` is not zero)

3.30 NatSpec Format

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

> **Note**
>
> NatSpec was inspired by Doxygen. While it uses Doxygen-style comments and tags, there is no intention to keep strict compatibility with Doxygen. Please carefully examine the supported tags listed below.

This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler. Also detailed below is output of the Solidity compiler, which extracts these comments into a machine-readable format.

NatSpec may also include annotations used by third-party tools. These are most likely accomplished via the `@custom:<name>` tag, and a good use case is analysis and verification tools.

3.30.1 Documentation Example

Documentation is inserted above each `contract`, `interface`, `library`, `function`, `enum`, `enum` value and `event` using the Doxygen notation format. A `public` state variable is equivalent to a `function` for the purposes of NatSpec.

- For Solidity you may choose `///` for single or multi-line comments, or `/**` and ending with `*/`.
- For Vyper, use `"""` indented to the inner contents with bare comments. See the Vyper documentation.

The following example shows a contract and a function using all available tags.

> **Note**
>
> The Solidity compiler only interprets tags if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.
>
> This may change in the future.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title A simulator for trees
```

(continues on next page)

(continued from previous page)

```
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
/// @custom:experimental This is an experimental contract.
contract Tree {
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return Age in years, rounded up for partial years
    /// @return Name of the tree
    function age(uint256 rings) external virtual pure returns (uint256, string memory) {
        return (rings + 1, "tree");
    }

    /// @notice Returns the amount of leaves the tree has.
    /// @dev Returns only a fixed number.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}

contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256, string memory) {
        return (rings + 2, "Kumquat");
    }

    /// Return the amount of leaves that this specific kind of tree has
    /// @inheritdoc Tree
    function leaves() external override(Tree, Plant) pure returns(uint256) {
        return 3;
    }
}
```

3.30.2 Tags

All tags are optional. The following table explains the purpose of each NatSpec tag and where it may be used. As a special case, if no tags are used then the Solidity compiler will interpret a `///` or `/**` comment in the same way as if it were tagged with `@notice`.

Tag		Context
`@title`	A title that should describe the contract/interface	contract, library, interface, struct, enum, enum values
`@author`	The name of the author	contract, library, interface, struct, enum, enum values
`@notice`	Explain to an end user what this does	contract, library, interface, function, public state variable, event, struct, enum, enum values error
`@dev`	Explain to a developer any extra details	contract, library, interface, function, state variable, event, struct, enum, enum values, error
`@param`	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event, enum values, error
`@return`	Documents the return variables of a contract's function	function, enum, enum values, public state variable
`@inheri`	Copies all missing tags from the base function (must be followed by the contract name)	function, enum, enum values, public state variable
`@custom ..`	Custom tag, semantics is application-defined	everywhere

If your function returns multiple values, like `(int quotient, int remainder)` then use multiple `@return` statements in the same format as the `@param` statements.

Custom tags start with `@custom:` and must be followed by one or more lowercase letters or hyphens. It cannot start with a hyphen however. They can be used everywhere and are part of the developer documentation.

Dynamic expressions

The Solidity compiler will pass through NatSpec documentation from your Solidity source code to the JSON output as described in this guide. The consumer of this JSON output, for example the end-user client software, may present this to the end-user directly or it may apply some pre-processing.

For example, some client software will render:

```
/// @notice This function will multiply `a` by 7
```

to the end-user as:

```
This function will multiply 10 by 7
```

if a function is being called and the input `a` is assigned a value of 10.

Inheritance Notes

Functions without NatSpec will automatically inherit the documentation of their base function. Exceptions to this are:

- When the parameter names are different.
- When there is more than one base function.
- When there is an explicit `@inheritdoc` tag which specifies which contract should be used to inherit.

3.30.3 Documentation Output

When parsed by the compiler, documentation such as the one from the above example will produce two different JSON files. One is meant to be consumed by the end user as a notice when a function is executed and the other to be used by the developer.

If the above contract is saved as `ex1.sol` then you can generate the documentation using:

```
solc --userdoc --devdoc ex1.sol
```

And the output is below.

> **Note**
>
> Starting Solidity version 0.6.11 the NatSpec output also contains a `version` and a `kind` field. Currently the `version` is set to `1` and `kind` must be one of `user` or `dev`. In the future it is possible that new versions will be introduced, deprecating older ones.

User Documentation

The above documentation will produce the following user documentation JSON file as output for the `Tree` contract:

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculate tree age in years, rounded up, for live trees"
    },
    "leaves()" :
    {
        "notice" : "Returns the amount of leaves the tree has."
    }
  },
  "notice" : "You can use this contract for only the most basic simulation"
}
```

Note that the key by which to find the methods is the function's canonical signature as defined in the *Contract ABI* and not simply the function's name.

Developer Documentation

Apart from the user documentation file, a developer documentation JSON file should also be produced and should look like this:

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "All function calls are currently implemented without side effects",
  "custom:experimental" : "This is an experimental contract.",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "The Alexandr N. Tetearing algorithm could increase precision",
      "params" :
      {
        "rings" : "The number of rings from dendrochronological sample"
```

(continues on next page)

(continued from previous page)

```
      },
      "returns" : {
        "_0" : "Age in years, rounded up for partial years",
        "_1" : "Name of the tree"
      }
    },
    "leaves()" :
    {
        "details" : "Returns only a fixed number."
    }
  },
  "title" : "A simulator for trees"
}
```

3.31 SMTChecker and Formal Verification

Using formal verification it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

Solidity implements a formal verification approach based on SMT (Satisfiability Modulo Theories) and Horn solving. The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

All the targets above are automatically checked by default if all engines are enabled, except underflow and overflow for Solidity >=0.8.7.

The potential warnings that the SMTChecker reports are:

- `<failing property> happens here.`. This means that the SMTChecker proved that a certain property fails. A counterexample may be given, however in complex situations it may also not show a counterexample. This result may also be a false positive in certain cases, when the SMT encoding adds abstractions for Solidity code that is either hard or impossible to express.
- `<failing property> might happen here`. This means that the solver could not prove either case within the given timeout. Since the result is unknown, the SMTChecker reports the potential failure for soundness. This may be solved by increasing the query timeout, but the problem might also simply be too hard for the engine to solve.

To enable the SMTChecker, you must select *which engine should run*, where the default is no engine. Selecting the engine enables the SMTChecker on all files.

Note

Prior to Solidity 0.8.4, the default way to enable the SMTChecker was via `pragma experimental SMTChecker;` and only the contracts containing the pragma would be analyzed. That pragma has been deprecated, and although it still enables the SMTChecker for backwards compatibility, it will be removed in Solidity 0.9.0. Note also that now using the pragma even in a single file enables the SMTChecker for all files.

Note

The lack of warnings for a verification target represents an undisputed mathematical proof of correctness, assuming no bugs in the SMTChecker and the underlying solver. Keep in mind that these problems are *very hard* and sometimes *impossible* to solve automatically in the general case. Therefore, several properties might not be solved or might lead to false positives for large contracts. Every proven property should be seen as an important achievement. For advanced users, see *SMTChecker Tuning* to learn a few options that might help proving more complex properties.

3.31.1 Tutorial

Overflow

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        return add(x, y);
    }
}
```

The contract above shows an overflow check example. The SMTChecker does not check underflow and overflow by default for Solidity >=0.8.7, so we need to use the command-line option `--model-checker-targets "underflow, overflow"` or the JSON option `settings.modelChecker.targets = ["underflow", "overflow"]`. See *this section for targets configuration*. Here, it reports the following:

```
Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
x = 1, y = 115792089237316195423570985008687907853269984665640564039457584007913129639935
```

(continues on next page)

(continued from previous page)

```
 = 0

Transaction trace:
Overflow.constructor(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935)
State: x = 1, y =␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935
Overflow.stateAdd()
    Overflow.add(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935) --␣
↪internal call
 --> o.sol:9:20:
  |
9 |             return x_ + y_;
  |                    ^^^^^^^
```

If we add `require` statements that filter out overflow cases, the SMTChecker proves that no overflow is reachable (by not reporting warnings):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        require(x < type(uint128).max);
        require(y < type(uint128).max);
        return add(x, y);
    }
}
```

Assert

An assertion represents an invariant in your code: a property that must be true *for all transactions, including all input and storage values*, otherwise there is a bug.

The code below defines a function `f` that guarantees no overflow. Function `inv` defines the specification that `f` is monotonically increasing: for every possible pair `(a, b)`, if `b > a` then `f(b) > f(a)`. Since `f` is indeed monotonically increasing, the SMTChecker proves that our property is correct. You are encouraged to play with the property and the function definition to see what results come out!

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;
```

(continues on next page)

(continued from previous page)

```
contract Monotonic {
    function f(uint x) internal pure returns (uint) {
        require(x < type(uint128).max);
        return x * 42;
    }

    function inv(uint a, uint b) public pure {
        require(b > a);
        assert(f(b) > f(a));
    }
}
```

We can also add assertions inside loops to verify more complicated properties. The following code searches for the maximum element of an unrestricted array of numbers, and asserts the property that the found element must be greater or equal every element in the array.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m >= a[i]);

        return m;
    }
}
```

Note that in this example the SMTChecker will automatically try to prove three properties:

1. `++i` in the first loop does not overflow.
2. `++i` in the second loop does not overflow.
3. The assertion is always true.

Note

The properties involve loops, which makes it *much much* harder than the previous examples, so beware of loops!

All the properties are correctly proven safe. Feel free to change the properties and/or add restrictions on the array to see different results. For example, changing the code to

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
```

(continues on next page)

(continued from previous page)

```
    function max(uint[] memory a) public pure returns (uint) {
        require(a.length >= 5);
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m > a[i]);

        return m;
    }
}
```

gives us:

```
Warning: CHC: Assertion violation happens here.
Counterexample:

a = [0, 0, 0, 0, 0]
 = 0

Transaction trace:
Test.constructor()
Test.max([0, 0, 0, 0, 0])
  --> max.sol:14:4:
   |
14 |            assert(m > a[i]);
```

State Properties

So far the examples only demonstrated the use of the SMTChecker over pure code, proving properties about specific operations or algorithms. A common type of properties in smart contracts are properties that involve the state of the contract. Multiple transactions might be needed to make an assertion fail for such a property.

As an example, consider a 2D grid where both axis have coordinates in the range (-2^127, 2^127 - 1). Let us place a robot at position (0, 0). The robot can only move diagonally, one step at a time, and cannot move outside the grid. The robot's state machine can be represented by the smart contract below.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Robot {
    int x = 0;
    int y = 0;

    modifier wall {
        require(x > type(int128).min && x < type(int128).max);
        require(y > type(int128).min && y < type(int128).max);
        _;
    }

    function moveLeftUp() wall public {
```

(continues on next page)

(continued from previous page)

```
        --x;
        ++y;
    }

    function moveLeftDown() wall public {
        --x;
        --y;
    }

    function moveRightUp() wall public {
        ++x;
        ++y;
    }

    function moveRightDown() wall public {
        ++x;
        --y;
    }

    function inv() public view {
        assert((x + y) % 2 == 0);
    }
}
```

Function `inv` represents an invariant of the state machine that `x + y` must be even. The SMTChecker manages to prove that regardless how many commands we give the robot, even if infinitely many, the invariant can *never* fail. The interested reader may want to prove that fact manually as well. Hint: this invariant is inductive.

We can also trick the SMTChecker into giving us a path to a certain position we think might be reachable. We can add the property that (2, 4) is *not* reachable, by adding the following function.

```
function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}
```

This property is false, and while proving that the property is false, the SMTChecker tells us exactly *how* to reach (2, 4):

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 2, y = 4

Transaction trace:
Robot.constructor()
State: x = 0, y = 0
Robot.moveLeftUp()
State: x = (- 1), y = 1
Robot.moveRightUp()
State: x = 0, y = 2
Robot.moveRightUp()
State: x = 1, y = 3
Robot.moveRightUp()
State: x = 2, y = 4
Robot.reach_2_4()
```

(continues on next page)

(continued from previous page)

```
  --> r.sol:35:4:
   |
35 |            assert(!(x == 2 && y == 4));
   |            ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Note that the path above is not necessarily deterministic, as there are other paths that could reach (2, 4). The choice of which path is shown might change depending on the used solver, its version, or just randomly.

External Calls and Reentrancy

Every external call is treated as a call to unknown code by the SMTChecker. The reasoning behind that is that even if the code of the called contract is available at compile time, there is no guarantee that the deployed contract will indeed be the same as the contract where the interface came from at compile time.

In some cases, it is possible to automatically infer properties over state variables that are still true even if the externally called code can do anything, including reenter the caller contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Unknown {
    function run() external;
}

contract Mutex {
    uint x;
    bool lock;

    Unknown immutable unknown;

    constructor(Unknown u) {
        require(address(u) != address(0));
        unknown = u;
    }

    modifier mutex {
        require(!lock);
        lock = true;
        _;
        lock = false;
    }

    function set(uint x_) mutex public {
        x = x_;
    }

    function run() mutex public {
        uint xPre = x;
        unknown.run();
        assert(xPre == x);
    }
}
```

The example above shows a contract that uses a mutex flag to forbid reentrancy. The solver is able to infer that when

unknown.run() is called, the contract is already "locked", so it would not be possible to change the value of x, regardless of what the unknown called code does.

If we "forget" to use the mutex modifier on function set, the SMTChecker is able to synthesize the behavior of the externally called code so that the assertion fails:

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1, lock = true, unknown = 1

Transaction trace:
Mutex.constructor(1)
State: x = 0, lock = false, unknown = 1
Mutex.run()
    unknown.run() -- untrusted external call, synthesized as:
        Mutex.set(1) -- reentrant call
  --> m.sol:32:3:
   |
32 |                assert(xPre == x);
   |                ^^^^^^^^^^^^^^^^
```

3.31.2 SMTChecker Options and Tuning

Timeout

The SMTChecker uses a hardcoded resource limit (rlimit) chosen per solver, which is not precisely related to time. We chose the rlimit option as the default because it gives more determinism guarantees than time inside the solver.

This options translates roughly to "a few seconds timeout" per query. Of course many properties are very complex and need a lot of time to be solved, where determinism does not matter. If the SMTChecker does not manage to solve the contract properties with the default rlimit, a timeout can be given in milliseconds via the CLI option --model-checker-timeout <time> or the JSON option settings.modelChecker.timeout=<time>, where 0 means no timeout.

Verification Targets

The types of verification targets created by the SMTChecker can also be customized via the CLI option --model-checker-target <targets> or the JSON option settings.modelChecker.targets=<targets>. In the CLI case, <targets> is a no-space-comma-separated list of one or more verification targets, and an array of one or more targets as strings in the JSON input. The keywords that represent the targets are:

- Assertions: assert.
- Arithmetic underflow: underflow.
- Arithmetic overflow: overflow.
- Division by zero: divByZero.
- Trivial conditions and unreachable code: constantCondition.
- Popping an empty array: popEmptyArray.
- Out of bounds array/fixed bytes index access: outOfBounds.
- Insufficient funds for a transfer: balance.
- All of the above: default (CLI only).

A common subset of targets might be, for example: --model-checker-targets assert,overflow.

All targets are checked by default, except underflow and overflow for Solidity >=0.8.7.

There is no precise heuristic on how and when to split verification targets, but it can be useful especially when dealing with large contracts.

Proved Targets

If there are any proved targets, the SMTChecker issues one warning per engine stating how many targets were proved. If the user wishes to see all the specific proved targets, the CLI option `--model-checker-show-proved-safe` and the JSON option `settings.modelChecker.showProvedSafe = true` can be used.

Unproved Targets

If there are any unproved targets, the SMTChecker issues one warning stating how many unproved targets there are. If the user wishes to see all the specific unproved targets, the CLI option `--model-checker-show-unproved` and the JSON option `settings.modelChecker.showUnproved = true` can be used.

Unsupported Language Features

Certain Solidity language features are not completely supported by the SMT encoding that the SMTChecker applies, for example assembly blocks. The unsupported construct is abstracted via overapproximation to preserve soundness, meaning any properties reported safe are safe even though this feature is unsupported. However such abstraction may cause false positives when the target properties depend on the precise behavior of the unsupported feature. If the encoder encounters such cases it will by default report a generic warning stating how many unsupported features it has seen. If the user wishes to see all the specific unsupported features, the CLI option `--model-checker-show-unsupported` and the JSON option `settings.modelChecker.showUnsupported = true` can be used, where their default value is `false`.

Verified Contracts

By default all the deployable contracts in the given sources are analyzed separately as the one that will be deployed. This means that if a contract has many direct and indirect inheritance parents, all of them will be analyzed on their own, even though only the most derived will be accessed directly on the blockchain. This causes an unnecessary burden on the SMTChecker and the solver. To aid cases like this, users can specify which contracts should be analyzed as the deployed one. The parent contracts are of course still analyzed, but only in the context of the most derived contract, reducing the complexity of the encoding and generated queries. Note that abstract contracts are by default not analyzed as the most derived by the SMTChecker.

The chosen contracts can be given via a comma-separated list (whitespace is not allowed) of <source>:<contract> pairs in the CLI: `--model-checker-contracts "<source1.sol:contract1>,<source2.sol:contract2>,<source2.sol:contract3>"`, and via the object `settings.modelChecker.contracts` in the *JSON input*, which has the following form:

```
"contracts": {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
}
```

Trusted External Calls

By default, the SMTChecker does not assume that compile-time available code is the same as the runtime code for external calls. Take the following contracts as an example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;
```

(continues on next page)

(continued from previous page)

```
contract Ext {
    uint public x;
    function setX(uint _x) public { x = _x; }
}
contract MyContract {
    function callExt(Ext _e) public {
        _e.setX(42);
        assert(_e.x() == 42);
    }
}
```

When `MyContract.callExt` is called, an address is given as the argument. At deployment time, we cannot know for sure that address `_e` actually contains a deployment of contract `Ext`. Therefore, the SMTChecker will warn that the assertion above can be violated, which is true, if `_e` contains another contract than `Ext`.

However, it can be useful to treat these external calls as trusted, for example, to test that different implementations of an interface conform to the same property. This means assuming that address `_e` indeed was deployed as contract `Ext`. This mode can be enabled via the CLI option `--model-checker-ext-calls=trusted` or the JSON field `settings.modelChecker.extCalls: "trusted"`.

Please be aware that enabling this mode can make the SMTChecker analysis much more computationally costly.

An important part of this mode is that it is applied to contract types and high level external calls to contracts, and not low level calls such as `call` and `delegatecall`. The storage of an address is stored per contract type, and the SMTChecker assumes that an externally called contract has the type of the caller expression. Therefore, casting an `address` or a contract to different contract types will yield different storage values and can give unsound results if the assumptions are inconsistent, such as the example below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract D {
    constructor(uint _x) { x = _x; }
    uint public x;
    function setX(uint _x) public { x = _x; }
}

contract E {
    constructor() { x = 2; }
    uint public x;
    function setX(uint _x) public { x = _x; }
}

contract C {
    function f() public {
        address d = address(new D(42));

        // `d` was deployed as `D`, so its `x` should be 42 now.
        assert(D(d).x() == 42); // should hold
        assert(D(d).x() == 43); // should fail

        // E and D have the same interface, so the following
        // call would also work at runtime.
        // However, the change to `E(d)` is not reflected in `D(d)`.
```

(continues on next page)

(continued from previous page)

```
        E(d).setX(1024);

        // Reading from `D(d)` now will show old values.
        // The assertion below should fail at runtime,
        // but succeeds in this mode's analysis (unsound).
        assert(D(d).x() == 42);
        // The assertion below should succeed at runtime,
        // but fails in this mode's analysis (false positive).
        assert(D(d).x() == 1024);
    }
}
```

Due to the above, make sure that the trusted external calls to a certain variable of `address` or `contract` type always have the same caller expression type.

It is also helpful to cast the called contract's variable as the type of the most derived type in case of inheritance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Token {
    function balanceOf(address _a) external view returns (uint);
    function transfer(address _to, uint _amt) external;
}

contract TokenCorrect is Token {
    mapping (address => uint) balance;
    constructor(address _a, uint _b) {
        balance[_a] = _b;
    }
    function balanceOf(address _a) public view override returns (uint) {
        return balance[_a];
    }
    function transfer(address _to, uint _amt) public override {
        require(balance[msg.sender] >= _amt);
        balance[msg.sender] -= _amt;
        balance[_to] += _amt;
    }
}

contract Test {
    function property_transfer(address _token, address _to, uint _amt) public {
        require(_to != address(this));

        TokenCorrect t = TokenCorrect(_token);

        uint xPre = t.balanceOf(address(this));
        require(xPre >= _amt);
        uint yPre = t.balanceOf(_to);

        t.transfer(_to, _amt);
        uint xPost = t.balanceOf(address(this));
        uint yPost = t.balanceOf(_to);
```

(continues on next page)

(continued from previous page)

```
        assert(xPost == xPre - _amt);
        assert(yPost == yPre + _amt);
    }
}
```

Note that in function `property_transfer`, the external calls are performed on variable `t`.

Another caveat of this mode are calls to state variables of contract type outside the analyzed contract. In the code below, even though `B` deploys `A`, it is also possible for the address stored in `B.a` to be called by anyone outside of `B` in between transactions to `B` itself. To reflect the possible changes to `B.a`, the encoding allows an unbounded number of calls to be made to `B.a` externally. The encoding will keep track of `B.a`'s storage, therefore assertion (2) should hold. However, currently the encoding allows such calls to be made from `B` conceptually, therefore assertion (3) fails. Making the encoding stronger logically is an extension of the trusted mode and is under development. Note that the encoding does not keep track of storage for `address` variables, therefore if `B.a` had type `address` the encoding would assume that its storage does not change in between transactions to `B`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract A {
    uint public x;
    address immutable public owner;
    constructor() {
        owner = msg.sender;
    }
    function setX(uint _x) public {
        require(msg.sender == owner);
        x = _x;
    }
}

contract B {
    A a;
    constructor() {
        a = new A();
        assert(a.x() == 0); // (1) should hold
    }
    function g() public view {
        assert(a.owner() == address(this)); // (2) should hold
        assert(a.x() == 0); // (3) should hold, but fails due to a false positive
    }
}
```

Reported Inferred Inductive Invariants

For properties that were proved safe with the CHC engine, the SMTChecker can retrieve inductive invariants that were inferred by the Horn solver as part of the proof. Currently only two types of invariants can be reported to the user:

- Contract Invariants: these are properties over the contract's state variables that are true before and after every possible transaction that the contract may ever run. For example, `x >= y`, where `x` and `y` are a contract's state variables.
- Reentrancy Properties: they represent the behavior of the contract in the presence of external calls to unknown

code. These properties can express a relation between the value of the state variables before and after the external call, where the external call is free to do anything, including making reentrant calls to the analyzed contract. Primed variables represent the state variables' values after said external call. Example: `lock -> x = x'`.

The user can choose the type of invariants to be reported using the CLI option `--model-checker-invariants "contract,reentrancy"` or as an array in the field `settings.modelChecker.invariants` in the *JSON input*. By default the SMTChecker does not report invariants.

Division and Modulo With Slack Variables

Spacer, the default Horn solver used by the SMTChecker, often dislikes division and modulo operations inside Horn rules. Because of that, by default the Solidity division and modulo operations are encoded using the constraint `a = b * d + m` where `d = a / b` and `m = a % b`. However, other solvers, such as Eldarica, prefer the syntactically precise operations. The command-line flag `--model-checker-div-mod-no-slacks` and the JSON option `settings.modelChecker.divModNoSlacks` can be used to toggle the encoding depending on the used solver preferences.

Natspec Function Abstraction

Certain functions including common math methods such as `pow` and `sqrt` may be too complex to be analyzed in a fully automated way. These functions can be annotated with Natspec tags that indicate to the SMTChecker that these functions should be abstracted. This means that the body of the function is not used, and when called, the function will:

- Return a nondeterministic value, and either keep the state variables unchanged if the abstracted function is view/pure, or also set the state variables to nondeterministic values otherwise. This can be used via the annotation `/// @custom:smtchecker abstract-function-nondet`.
- Act as an uninterpreted function. This means that the semantics of the function (given by the body) are ignored, and the only property this function has is that given the same input it guarantees the same output. This is currently under development and will be available via the annotation `/// @custom:smtchecker abstract-function-uf`.

Model Checking Engines

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.

By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc,chc,none}` or the JSON option `settings.modelChecker.engine={all,bmc,chc,none}`.

Bounded Model Checker (BMC)

Warning

The BMC engine has been deprecated and will be removed in a future release. Selecting it, either explicitly via `bmc` or implicitly via `all`, emits a deprecation warning. Please use the CHC engine instead.

The BMC engine analyzes functions in isolation, that is, it does not take the overall behavior of the contract over multiple transactions into account when analyzing each function. Loops are also ignored in this engine at the moment. Internal function calls are inlined as long as they are not recursive, directly or indirectly. External function calls are inlined if possible. Knowledge that is potentially affected by reentrancy is erased.

The characteristics above make BMC prone to reporting false positives, but it is also lightweight and should be able to quickly find small local bugs.

Constrained Horn Clauses (CHC)

A contract's Control Flow Graph (CFG) is modelled as a system of Horn clauses, where the life cycle of the contract is represented by a loop that can visit every public/external function non-deterministically. This way, the behavior of the entire contract over an unbounded number of transactions is taken into account when analyzing any function. Loops are fully supported by this engine. Internal function calls are supported, and external function calls assume the called code is unknown and can do anything.

The CHC engine is much more powerful than BMC in terms of what it can prove, and might require more computing resources.

SMT and Horn solvers

The two engines detailed above use automated theorem provers as their logical backends. BMC uses an SMT solver, whereas CHC uses a Horn solver. Often the same tool can act as both, as seen in z3, which is primarily an SMT solver and makes Spacer available as a Horn solver, and Eldarica which does both.

The user can choose which solvers should be used, if available, via the CLI option `--model-checker-solvers {all,cvc5,eld,smtlib2,z3}` or the JSON option `settings.modelChecker.solvers=[smtlib2,z3]`, where:

- `cvc5` is used via its binary which must be installed in the system. Only BMC uses `cvc5`.
- `eld` is used via its binary which must be installed in the system. Only CHC uses `eld`, and only if `z3` is not enabled.
- `smtlib2` outputs SMT/Horn queries in the smtlib2 format. These can be used together with the compiler's callback mechanism so that any solver binary from the system can be employed to synchronously return the results of the queries to the compiler. This can be used by both BMC and CHC depending on which solvers are called.
- `z3` is available statically in `soljson.js` (from Solidity 0.6.9), that is, the JavaScript binary of the compiler. Otherwise it is used via its binary which must be installed in the system.

Note

z3 version 4.8.16 broke ABI compatibility with previous versions and cannot be used with solc <=0.8.13. If you are using z3 >=4.8.16 please use solc >=0.8.14, and conversely, only use older z3 with older solc releases. We also recommend using the latest z3 release which is what SMTChecker also does.

Since both BMC and CHC use `z3`, and `z3` is available in a greater variety of environments, including in the browser, most users will almost never need to be concerned about this option. More advanced users might apply this option to try alternative solvers on more complex problems.

Please note that certain combinations of chosen engine and solver will lead to the SMTChecker doing nothing, for example choosing CHC and `cvc5`.

3.31.3 Abstraction and False Positives

The SMTChecker implements abstractions in an incomplete and sound way: If a bug is reported, it might be a false positive introduced by abstractions (due to erasing knowledge or using a non-precise type). If it determines that a verification target is safe, it is indeed safe, that is, there are no false negatives (unless there is a bug in the SMTChecker).

If a target cannot be proven you can try to help the solver by using the tuning options in the previous section. If you are sure of a false positive, adding `require` statements in the code with more information may also give some more power to the solver.

SMT Encoding and Types

The SMTChecker encoding tries to be as precise as possible, mapping Solidity types and expressions to their closest SMT-LIB representation, as shown in the table below.

Solidity type	SMT sort	Theories
Boolean	Bool	Bool
intN, uintN, address, bytesN, enum, contract	Integer	LIA, NIA
array, mapping, bytes, string	Tuple (Array elements, Integer length)	Datatypes, Arrays, LIA
struct	Tuple	Datatypes
other types	Integer	LIA

Types that are not yet supported are abstracted by a single 256-bit unsigned integer, where their unsupported operations are ignored.

For more details on how the SMT encoding works internally, see the paper SMT-based Verification of Solidity Smart Contracts.

Function Calls

In the BMC engine, function calls to the same contract (or base contracts) are inlined when possible, that is, when their implementation is available. Calls to functions in other contracts are not inlined even if their code is available, since we cannot guarantee that the actual deployed code is the same.

The CHC engine creates nonlinear Horn clauses that use summaries of the called functions to support internal function calls. External function calls are treated as calls to unknown code, including potential reentrant calls.

Complex pure functions are abstracted by an uninterpreted function (UF) over the arguments.

Functions	BMC/CHC behavior
`assert`	Verification target.
`require`	Assumption.
internal call	BMC: Inline function call. CHC: Function summaries.
external call to known code	BMC: Inline function call or erase knowledge about state variables and local storage references. CHC: Assume called code is unknown. Try to infer invariants that hold after the call returns.
Storage array push/pop	Supported precisely. Checks whether it is popping an empty array.
ABI functions	Abstracted with UF.
`addmod`, `mulmod`	Supported precisely.
`gasleft`, `blobhash`, `blockhash`, `keccak256`, `ecrecover`, `ripemd160`	Abstracted with UF.
pure functions without implementation (external or complex)	Abstracted with UF
external functions without implementation	BMC: Erase state knowledge and assume result is nondeterministic. CHC: Nondeterministic summary. Try to infer invariants that hold after the call returns.
transfer	BMC: Checks whether the contract's balance is sufficient. CHC: does not yet perform the check.
others	Currently unsupported

Using abstraction means loss of precise knowledge, but in many cases it does not mean loss of proving power.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Recover
{
    function f(
        bytes32 hash,
        uint8 v1, uint8 v2,
        bytes32 r1, bytes32 r2,
        bytes32 s1, bytes32 s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, v1, r1, s1);
        require(v1 == v2);
        require(r1 == r2);
        require(s1 == s2);
        address a2 = ecrecover(hash, v2, r2, s2);
        assert(a1 == a2);
        return a1;
    }
}
```

In the example above, the SMTChecker is not expressive enough to actually compute `ecrecover`, but by modelling the function calls as uninterpreted functions we know that the return value is the same when called on equivalent parameters. This is enough to prove that the assertion above is always true.

Abstracting a function call with an UF can be done for functions known to be deterministic, and can be easily done for pure functions. It is however difficult to do this with general external functions, since they might depend on state variables.

Reference Types and Aliasing

Solidity implements aliasing for reference types with the same *data location*. That means one variable may be modified through a reference to the same data area. The SMTChecker does not keep track of which references refer to the same data. This implies that whenever a local reference or state variable of reference type is assigned, all knowledge regarding variables of the same type and data location is erased. If the type is nested, the knowledge removal also includes all the prefix base types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Aliasing
{
    uint[] array1;
    uint[][] array2;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal {
        array1[0] = 42;
        a[0] = 2;
        c[0][0] = 2;
        b[0] = 1;
```

(continues on next page)

(continued from previous page)

```
        // Erasing knowledge about memory references should not
        // erase knowledge about state variables.
        assert(array1[0] == 42);
        // However, an assignment to a storage reference will erase
        // storage knowledge accordingly.
        d[0] = 2;
        // Fails as false positive because of the assignment above.
        assert(array1[0] == 42);
        // Fails because `a == b` is possible.
        assert(a[0] == 2);
        // Fails because `c[i] == b` is possible.
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
    function g(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint x
    ) public {
        f(a, b, c, array2[x]);
    }
}
```

After the assignment to `b[0]`, we need to clear knowledge about `a` since it has the same type (`uint[]`) and data location (memory). We also need to clear knowledge about `c`, since its base type is also a `uint[]` located in memory. This implies that some `c[i]` could refer to the same data as `b` or `a`.

Notice that we do not clear knowledge about `array` and `d` because they are located in storage, even though they also have type `uint[]`. However, if `d` was assigned, we would need to clear knowledge about `array` and vice-versa.

Contract Balance

A contract may be deployed with funds sent to it, if `msg.value` > 0 in the deployment transaction. However, the contract's address may already have funds before deployment, which are kept by the contract. Therefore, the SMTChecker assumes that `address(this).balance >= msg.value` in the constructor in order to be consistent with the EVM rules. The contract's balance may also increase without triggering any calls to the contract, if

- `selfdestruct` is executed by another contract with the analyzed contract as the target of the remaining funds,
- the contract is the coinbase (i.e., `block.coinbase`) of some block.

To model this properly, the SMTChecker assumes that at every new transaction the contract's balance may grow by at least `msg.value`.

3.31.4 Real World Assumptions

Some scenarios can be expressed in Solidity and the EVM, but are expected to never occur in practice. One of such cases is the length of a dynamic storage array overflowing during a push: If the `push` operation is applied to an array of length 2^256 - 1, its length silently overflows. However, this is unlikely to happen in practice, since the operations required to grow the array to that point would take billions of years to execute. Another similar assumption taken by the SMTChecker is that an address' balance can never overflow.

A similar idea was presented in EIP-1985.

3.32 Yul

Yul (previously also called JULIA or IULIA) is an intermediate language that can be compiled to bytecode for different backends.

It can be used in stand-alone mode and for "inline assembly" inside Solidity. The compiler uses Yul as an intermediate language in the IR-based code generator ("new codegen" or "IR-based codegen"). Yul is a good target for high-level optimisation stages that can benefit all target platforms equally.

3.32.1 Motivation and High-level Description

The design of Yul tries to achieve several goals:

1. Programs written in Yul should be readable, even if the code is generated by a compiler from Solidity or another high-level language.
2. Control flow should be easy to understand to help in manual inspection, formal verification and optimization.
3. The translation from Yul to bytecode should be as straightforward as possible.
4. Yul should be suitable for whole-program optimization.

In order to achieve the first and second goal, Yul provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. These should be sufficient for adequately representing the control flow for assembly programs. Therefore, no explicit statements for `SWAP`, `DUP`, `JUMPDEST`, `JUMP` and `JUMPI` are provided, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

Even though it was designed for stack machines, Yul does not expose the complexity of the stack itself. The programmer or auditor should not have to worry about the stack.

The third goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...) and cleanup of local variables from the stack.

To avoid confusions between concepts like values and references, Yul is statically typed. At the same time, there is a default type (usually the integer word of the target machine) that can always be omitted to help readability.

To keep the language simple and flexible, Yul does not have any built-in operations, functions or types in its pure form. These are added together with their semantics when specifying a dialect of Yul, which allows specializing Yul to the requirements of different target platforms and feature sets.

Currently, there is only one specified dialect of Yul. This dialect uses the EVM opcodes as builtin functions (see below) and defines only the type `u256`, which is the native 256-bit type of the EVM. Because of that, we will not provide types in the examples below.

3.32.2 Simple Example

The following example program is written in the EVM dialect and computes exponentiation. It can be compiled using `solc --strict-assembly`. The builtin functions `mul` and `div` compute product and division, respectively.

```
{
    function power(base, exponent) -> result
    {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default
```

(continues on next page)

(continued from previous page)

```
        {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

It is also possible to implement the same function using a for-loop instead of with recursion. Here, `lt(a, b)` computes whether `a` is less than `b`.

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

At the *end of the section*, a complete implementation of the ERC-20 standard can be found.

3.32.3 Stand-Alone Usage

You can use Yul in its stand-alone form in the EVM dialect using the Solidity compiler. This will use the *Yul object notation* so that it is possible to refer to code as data to deploy contracts. This Yul mode is available for the commandline compiler (use `--strict-assembly`) and for the *standard-json interface*:

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
    "settings": {
        "outputSelection": { "*": { "*": ["*"], "": [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

Warning

Yul is in active development and bytecode generation is only fully implemented for the EVM dialect of Yul with EVM 1.0 as target.

3.32.4 Informal Description of Yul

In the following, we will talk about each individual aspect of the Yul language. In examples, we will use the default EVM dialect.

Syntax

Yul parses comments, literals and identifiers in the same way as Solidity, so you can e.g. use `//` and `/* */` to denote comments. There is one exception: Identifiers in Yul can contain dots: `..`

Yul can specify "objects" that consist of code, data and sub-objects. Please see *Yul Objects* below for details on that. In this section, we are only concerned with the code part of such an object. This code part always consists of a curly-braces delimited block. Most tools support specifying just a code block where an object is expected.

Inside a code block, the following elements can be used (see the later sections for more details):

- literals, e.g. `0x123`, `42` or `"abc"` (strings up to 32 characters)
- calls to builtin functions, e.g. `add(1, mload(0))`
- variable declarations, e.g. `let x := 7`, `let x := add(y, 3)` or `let x` (initial value of 0 is assigned)
- identifiers (variables), e.g. `add(3, x)`
- assignments, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`
- if statements, e.g. `if lt(a, b) { sstore(0, 1) }`
- switch statements, e.g. `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- for loops, e.g. `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- function definitions, e.g. `function f(a, b) -> c { c := add(a, b) }`

Multiple syntactical elements can follow each other simply separated by whitespace, i.e. there is no terminating `;` or newline required.

Literals

As literals, you can use:

- Integer constants in decimal or hexadecimal notation.
- ASCII strings (e.g. `"abc"`), which may contain hex escapes `\xNN` and Unicode escapes `\uNNNN` where `N` are hexadecimal digits.
- Hex strings (e.g. `hex"616263"`).

In the EVM dialect of Yul, literals represent 256-bit words as follows:

- Decimal or hexadecimal constants must be less than `2**256`. They represent the 256-bit word with that value as an unsigned integer in big endian encoding.
- An ASCII string is first viewed as a byte sequence, by viewing a non-escape ASCII character as a single byte whose value is the ASCII code, an escape `\xNN` as single byte with that value, and an escape `\uNNNN` as the UTF-8 sequence of bytes for that code point. The byte sequence must not exceed 32 bytes. The byte sequence is padded with zeros on the right to reach 32 bytes in length; in other words, the string is stored left-aligned. The padded byte sequence represents a 256-bit word whose most significant 8 bits are the ones from the first byte, i.e. the bytes are interpreted in big endian form.
- A hex string is first viewed as a byte sequence, by viewing each pair of contiguous hex digits as a byte. The byte sequence must not exceed 32 bytes (i.e. 64 hex digits), and is treated as above.

When compiling for the EVM, this will be translated into an appropriate `PUSHi` instruction. In the following example, `3` and `2` are added resulting in 5 and then the bitwise `and` with the string "abc" is computed. The final value is assigned to a local variable called `x`.

The 32-byte limit above does not apply to string literals passed to builtin functions that require literal arguments (e.g. `setimmutable` or `loadimmutable`). Those strings never end up in the generated bytecode.

```
let x := and("abc", add(3, 2))
```

Unless it is the default type, the type of a literal has to be specified after a colon:

```
// This will not compile (u32 and u256 type not implemented yet)
let x := and("abc":u32, add(3:u256, 2:u256))
```

Function Calls

Both built-in and user-defined functions (see below) can be called in the same way as shown in the previous example. If the function returns a single value, it can be directly used inside an expression again. If it returns multiple values, they have to be assigned to local variables.

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// Here, the user-defined function `f` returns two values.
let x, y := f(1, mload(0))
```

For built-in functions of the EVM, functional expressions can be directly translated to a stream of opcodes: You just read the expression from right to left to obtain the opcodes. In the case of the second line in the example, this is `PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE`.

For calls to user-defined functions, the arguments are also put on the stack from right to left and this is the order in which argument lists are evaluated. The return values, though, are expected on the stack from left to right, i.e. in this example, `y` is on top of the stack and `x` is below it.

Variable Declarations

You can use the `let` keyword to declare variables. A variable is only visible inside the `{...}`-block it was defined in. When compiling to the EVM, a new stack slot is created that is reserved for the variable and automatically removed again when the end of the block is reached. You can provide an initial value for the variable. If you do not provide a value, the variable will be initialized to zero.

Since variables are stored on the stack, they do not directly influence memory or storage, but they can be used as pointers to memory or storage locations in the built-in functions `mstore`, `mload`, `sstore` and `sload`. Future dialects might introduce specific types for such pointers.

When a variable is referenced, its current value is copied. For the EVM, this translates to a `DUP` instruction.

```
{
    let zero := 0
    let v := calldataload(zero)
    {
        let y := add(sload(v), 1)
        v := y
    } // y is "deallocated" here
    sstore(v, zero)
} // v and zero are "deallocated" here
```

If the declared variable should have a type different from the default type, you denote that following a colon. You can also declare multiple variables in one statement when you assign from a function call that returns multiple values.

```
// This will not compile (u32 and u256 type not implemented yet)
{
    let zero:u32 := 0:u32
```

(continues on next page)

(continued from previous page)

```
    let v:u256, t:u32 := f()
    let x, y := g()
}
```

Depending on the optimiser settings, the compiler can free the stack slots already after the variable has been used for the last time, even though it is still in scope.

Assignments

Variables can be assigned to after their definition using the `:=` operator. It is possible to assign multiple variables at the same time. For this, the number and types of the values have to match. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables. The same variable may not occur multiple times on the left-hand side of an assignment, e.g. `x, x := f()` is invalid.

```
let v := 0
// re-assign v
v := 2
let t := add(v, 2)
function f() -> a, b { }
// assign multiple values
v, t := f()
```

If

The if statement can be used for conditionally executing code. No "else" block can be defined. Consider using "switch" instead (see below) if you need multiple alternatives.

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

The curly braces for the body are required.

Switch

You can use a switch statement as an extended version of the if statement. It takes the value of an expression and compares it to several literal constants. The branch corresponding to the matching constant is taken. Contrary to other programming languages, for safety reasons, control flow does not continue from one case to the next. There can be a fallback or default case called `default` which is taken if none of the literal constants matches.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases is not enclosed by curly braces, but the body of a case does require them.

Loops

Yul supports for-loops which consist of a header containing an initializing part, a condition, a post-iteration part and a body. The condition has to be an expression, while the other three are blocks. If the initializing part declares any variables at the top level, the scope of these variables extends to all other parts of the loop.

The `break` and `continue` statements can be used in the body to exit the loop or skip to the post-part, respectively.

The following example computes the sum of an area in memory.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For loops can also be used as a replacement for while loops: Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {     // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Function Declarations

Yul allows the definition of functions. These should not be confused with functions in Solidity since they are never part of an external interface of a contract and are part of a namespace separate from the one for Solidity functions.

For the EVM, Yul functions take their arguments (and a return PC) from the stack and also put the results onto the stack. User-defined functions and built-in functions are called in exactly the same way.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function.

Functions declare parameters and return variables, similar to Solidity. To return a value, you assign it to the return variable(s).

If you call a function that returns multiple values, you have to assign them to multiple variables using `a, b := f(x)` or `let a, b := f(x)`.

The `leave` statement can be used to exit the current function. It works like the `return` statement in other languages just that it does not take a value to return, it just exits the functions and the function will return whatever values are currently assigned to the return variable(s).

Note that the EVM dialect has a built-in function called `return` that quits the full execution context (internal message call) and not just the current yul function.

The following example implements the power function by square-and-multiply.

```
{
    function power(base, exponent) -> result {
        switch exponent
        case 0 { result := 1 }
```

(continues on next page)

(continued from previous page)

```
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

3.32.5 Specification of Yul

This chapter describes Yul code formally. Yul code is usually placed inside Yul objects, which are explained in their own chapter.

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9.]*
IdentifierList = Identifier ( ',' Identifier)*
```

(continues on next page)

(continued from previous page)

```
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Restrictions on the Grammar

Apart from those directly imposed by the grammar, the following restrictions apply:

Switches must have at least one case (including the default case). All case values need to have the same type and distinct values. If all possible values of the expression type are covered, a default case is not allowed (i.e. a switch with a `bool` expression that has both a true and a false case do not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return variables of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed. The same variable name cannot occur more than once in the left-hand-side of an assignment or variable declaration.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

A `continue` or `break` statement can only be used inside the body of a for-loop, as follows. Consider the innermost loop that contains the statement. The loop and the statement must be in the same function, or both must be at the top level. The statement must be in the loop's body block; it cannot be in the loop's initialization block or update block. It is worth emphasizing that this restriction applies just to the innermost loop that contains the `continue` or `break` statement: this innermost loop, and therefore the `continue` or `break` statement, may appear anywhere in an outer loop, possibly in an outer loop's initialization block or update block. For example, the following is legal, because the `break` occurs in the body block of the inner loop, despite also occurring in the update block of the outer loop:

```
for {} true { for {} true {} { break } }
{
}
```

The condition part of the for-loop has to evaluate to exactly one value.

The `leave` statement can only be used inside a function.

Functions cannot be defined anywhere inside for loop init blocks.

Literals cannot be larger than their type. The largest type defined is 256-bit wide.

During assignments and function calls, the types of the respective values have to match. There is no implicit type conversion. Type conversion in general can only be achieved if the dialect provides an appropriate built-in function that takes a value of one type and returns a value of a different type.

Scoping Rules

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations (`FunctionDefinition`, `VariableDeclaration`) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks): Functions are visible in the whole block (even before their definitions) while variables are only visible starting from the statement after the `VariableDeclaration`.

In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

As an exception to the general scoping rule, the scope of the "init" part of the for-loop (the first block) extends across all other parts of the for loop. This means that variables (and functions) declared in the init part (but not inside a block inside the init part) are visible in all other parts of the for-loop.

Identifiers declared in the other parts of the for loop respect the regular syntactical scoping rules.

This means a for-loop of the form `for { I... } C { P... } { B... }` is equivalent to `{ I... for {} C { P... } { B... } }`.

The parameters and return parameters of functions are visible in the function body and their names have to be distinct.

Inside functions, it is not possible to reference a variable that was declared outside of that function.

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not possible to reference it because it was declared outside the current function.

Formal Specification

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. As builtin functions can have side effects, E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM).

If the AST node is a statement, E returns the two state objects and a "mode", which is used for the `break`, `continue` and `leave` statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state `L` is a mapping of identifiers `i` to values `v`, denoted as `L[i] = v`.

For an identifier `v`, let `$v` be the name of the identifier.

We will use a destructuring notation for the AST nodes.

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise
            G1, L1, mode
```

(continues on next page)

(continued from previous page)

```
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
    G1, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L1, mode = E(G, L, i1, ..., in)
        // mode has to be regular or leave due to the syntactic restrictions
        if mode is leave then
            G1, L1 restricted to variables of L, leave
        otherwise
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 restricted to variables of L, mode
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G3, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
```

(continues on next page)

(continued from previous page)

```
    // i = 1 .. n
    // Evaluate literals, context doesn't matter
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
    Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, L', block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: StringLiteral) = G, L, str(l),
    where str is the string evaluation function,
    which for the EVM dialect is defined in the section 'Literals' above
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal evaluation function,
    which turns a sequence of hexadecimal digits into their big endian value
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal evaluation function,
    which turns a sequence of decimal digits into their big endian value
```

EVM Dialect

The default dialect of Yul currently is the EVM dialect for the currently selected version of the EVM. The only type available in this dialect is `u256`, the 256-bit native type of the Ethereum Virtual Machine. Since it is the default type of this dialect, it can be omitted.

The following table lists all builtin functions (depending on the EVM version) and provides a short description of the semantics of the function / opcode. This document does not want to be a full description of the Ethereum virtual machine. Please refer to a different document if you are interested in the precise semantics.

Opcodes marked with `-` do not return a result and all others return exactly one value. Opcodes marked with `F`, `H`, `B`, `C`, `I`, `L`, `P`, `N` and `O` are present since Frontier, Homestead, Byzantium, Constantinople, Istanbul, London, Paris, Cancun or Osaka respectively.

In the following, `mem[a...b)` signifies the bytes of memory starting at position `a` up to but not including position `b`, `storage[p]` signifies the storage contents at slot `p`, and similarly, `transientStorage[p]` signifies the transient storage contents at slot `p`.

Since Yul manages local variables and control-flow, opcodes that interfere with these features are not available. This includes the `dup` and `swap` instructions as well as `jump` instructions, labels and the `push` instructions.

Instruction			Explanation
stop()	-	F	stop execution, identical to return(0, 0)
add(x, y)		F	x + y
sub(x, y)		F	x - y
mul(x, y)		F	x * y
div(x, y)		F	x / y or 0 if y == 0
sdiv(x, y)		F	x / y, for signed numbers in two's complement, 0 if y == 0
mod(x, y)		F	x % y, 0 if y == 0
smod(x, y)		F	x % y, for signed numbers in two's complement, 0 if y == 0
exp(x, y)		F	x to the power of y
not(x)		F	bitwise "not" of x (every bit of x is negated)
lt(x, y)		F	1 if x < y, 0 otherwise
gt(x, y)		F	1 if x > y, 0 otherwise
slt(x, y)		F	1 if x < y, 0 otherwise, for signed numbers in two's complement
sgt(x, y)		F	1 if x > y, 0 otherwise, for signed numbers in two's complement
eq(x, y)		F	1 if x == y, 0 otherwise
iszero(x)		F	1 if x == 0, 0 otherwise
and(x, y)		F	bitwise "and" of x and y
or(x, y)		F	bitwise "or" of x and y
xor(x, y)		F	bitwise "xor" of x and y
byte(n, x)		F	nth byte of x, where the most significant byte is the 0th byte
shl(x, y)		C	logical shift left y by x bits
shr(x, y)		C	logical shift right y by x bits
sar(x, y)		C	signed arithmetic shift right y by x bits
clz(x)		O	number of leading zero bits of x, 256 if x == 0
addmod(x, y, m)		F	(x + y) % m with arbitrary precision arithmetic, 0 if m == 0
mulmod(x, y, m)		F	(x * y) % m with arbitrary precision arithmetic, 0 if m == 0
signextend(i, x)		F	sign extend from (i*8+7)th bit counting from least significant
keccak256(p, n)		F	keccak(mem[p…(p+n)))
pop(x)	-	F	discard value x
mload(p)		F	mem[p…(p+32))
mstore(p, v)	-	F	mem[p…(p+32)) := v
mstore8(p, v)	-	F	mem[p] := v & 0xff (only modifies a single byte)
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
tload(p)		N	transientStorage[p]
tstore(p, v)	-	N	transientStorage[p] := v
msize()		F	size of memory, i.e. largest accessed memory index
gas()		F	gas still available to execution
address()		F	address of the current contract / execution context
balance(a)		F	wei balance at address a
selfbalance()		I	equivalent to balance(address()), but cheaper
caller()		F	call sender (excluding `delegatecall`)
callvalue()		F	wei sent together with the current call
calldataload(p)		F	call data starting from position p (32 bytes)
calldatasize()		F	size of call data in bytes
calldatacopy(t, f, s)	-	F	copy s bytes from calldata at position f to mem at position t
codesize()		F	size of the code of the current contract / execution context
codecopy(t, f, s)	-	F	copy s bytes from code at position f to mem at position t
extcodesize(a)		F	size of the code at address a
extcodecopy(a, t, f, s)	-	F	like codecopy(t, f, s) but take code at address a
returndatasize()		B	size of the last returndata

Instruction			Explanation
returndatacopy(t, f, s)	-	B	copy s bytes from returndata at position f to mem at position t
mcopy(t, f, s)	-	N	copy s bytes from mem at position f to mem at position t
extcodehash(a)		C	code hash of address a
create(v, p, n)		F	create new contract with code mem[p…(p+n)) and send v wei and return the new
create2(v, p, n, s)		C	create new contract with code mem[p…(p+n)) at address keccak256(0xff . this . s
call(g, a, v, in, insize, out, outsize)		F	call contract at address a with input mem[in…(in+insize)) providing g gas and v w
callcode(g, a, v, in, insize, out, outsize)		F	identical to `call` but only use the code from a and stay in the context of the curren
delegatecall(g, a, in, insize, out, outsize)		H	identical to `callcode` but also keep `caller` and `callvalue` *See more*
staticcall(g, a, in, insize, out, outsize)		B	identical to `call(g, a, 0, in, insize, out, outsize)` but do not allow st
return(p, s)	-	F	end execution, return data mem[p…(p+s))
revert(p, s)	-	B	end execution, revert state changes, return data mem[p…(p+s))
selfdestruct(a)	-	F	end execution, destroy current contract and send funds to a (deprecated)
invalid()	-	F	end execution with invalid instruction
log0(p, s)	-	F	log data mem[p…(p+s))
log1(p, s, t1)	-	F	log data mem[p…(p+s)) with topic t1
log2(p, s, t1, t2)	-	F	log data mem[p…(p+s)) with topics t1, t2
log3(p, s, t1, t2, t3)	-	F	log data mem[p…(p+s)) with topics t1, t2, t3
log4(p, s, t1, t2, t3, t4)	-	F	log data mem[p…(p+s)) with topics t1, t2, t3, t4
chainid()		I	ID of the executing chain (EIP-1344)
basefee()		L	current block's base fee (EIP-3198 and EIP-1559)
blobbasefee()		N	current block's blob base fee (EIP-7516 and EIP-4844)
origin()		F	transaction sender
gasprice()		F	gas price of the transaction
blockhash(b)		F	hash of block nr b - only for last 256 blocks excluding current
blobhash(i)		N	versioned hash of transaction's i-th blob, 0 if blob does not exist
coinbase()		F	current mining beneficiary
timestamp()		F	timestamp of the current block in seconds since the epoch
number()		F	current block number
difficulty()		F	difficulty of the current block (see note below)
prevrandao()		P	randomness provided by the beacon chain (see note below)
gaslimit()		F	block gas limit of the current block

Note

The `call*` instructions use the `out` and `outsize` parameters to define an area in memory where the return or failure data is placed. This area is written to depending on how many bytes the called contract returns. If it returns more data, only the first `outsize` bytes are written. You can access the rest of the data using the `returndatacopy` opcode. If it returns less data, then the remaining bytes are not touched at all. You need to use the `returndatasize` opcode to check which part of this memory area contains the return data. The remaining bytes will retain their values as of before the call.

Note

The `difficulty()` instruction is disallowed in EVM version >= Paris. With the Paris network upgrade the semantics of the instruction that was previously called `difficulty` have been changed and the instruction was renamed to `prevrandao`. It can now return arbitrary values in the full 256-bit range, whereas the highest recorded difficulty value within Ethash was ~54 bits. This change is described in EIP-4399. Please note that irrelevant to which EVM version is selected in the compiler, the semantics of instructions depend on the final chain of deployment.

Warning

From version 0.8.18 and up, the use of `selfdestruct` in both Solidity and Yul will trigger a deprecation warning, since the `SELFDESTRUCT` opcode will eventually undergo breaking changes in behavior as stated in EIP-6049.

In some internal dialects, there are additional functions:

datasize, dataoffset, datacopy

The functions `datasize(x)`, `dataoffset(x)` and `datacopy(t, f, l)` are used to access other parts of a Yul object.

`datasize` and `dataoffset` can only take string literals (the names of other objects) as arguments and return the size and offset in the data area, respectively. For the EVM, the `datacopy` function is equivalent to `codecopy`.

setimmutable, loadimmutable

The functions `setimmutable(offset, "name", value)` and `loadimmutable("name")` are used for the immutable mechanism in Solidity and do not nicely map to pure Yul. The call to `setimmutable(offset, "name", value)` assumes that the runtime code of the contract containing the given named immutable was copied to memory at offset `offset` and will write `value` to all positions in memory (relative to `offset`) that contain the placeholder that was generated for calls to `loadimmutable("name")` in the runtime code.

linkersymbol

The function `linkersymbol("library_id")` is a placeholder for an address literal to be substituted by the linker. Its first and only argument must be a string literal and uniquely represents the address to be inserted. Identifiers can be arbitrary but when the compiler produces Yul code from Solidity sources, it uses a library name qualified with the name of the source unit that defines that library. To link the code with a particular library address, the same identifier must be provided to the `--libraries` option on the command-line.

For example this code

```
let a := linkersymbol("file.sol:Math")
```

is equivalent to

```
let a := 0x1234567890123456789012345678901234567890
```

when the linker is invoked with `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890` option.

See *Using the Commandline Compiler* for details about the Solidity linker.

memoryguard

This function is available in the EVM dialect with objects. The caller of `let ptr := memoryguard(size)` (where `size` has to be a literal number) promises that they only use memory in either the range `[0, size)` or the unbounded range starting at `ptr`.

Since the presence of a `memoryguard` call indicates that all memory access adheres to this restriction, it allows the optimizer to perform additional optimization steps, for example the stack limit evader, which attempts to move stack variables that would otherwise be unreachable to memory.

The Yul optimizer promises to only use the memory range `[size, ptr)` for its purposes. If the optimizer does not need to reserve any memory, it holds that `ptr == size`.

`memoryguard` can be called multiple times, but needs to have the same literal as argument within one Yul subobject. If at least one `memoryguard` call is found in a subobject, the additional optimiser steps will be run on it.

verbatim

The set of `verbatim...` builtin functions lets you create bytecode for opcodes that are not known to the Yul compiler. It also allows you to create bytecode sequences that will not be modified by the optimizer.

The functions are `verbatim_<n>i_<m>o("<data>", ...)`, where

- `n` is a decimal between 0 and 99 that specifies the number of input stack slots / variables
- `m` is a decimal between 0 and 99 that specifies the number of output stack slots / variables
- `data` is a string literal that contains the sequence of bytes

If you for example want to define a function that multiplies the input by two, without the optimizer touching the constant two, you can use

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

This code will result in a `dup1` opcode to retrieve `x` (the optimizer might directly reuse result of the `calldataload` opcode, though) directly followed by `600202`. The code is assumed to consume the copied value of `x` and produce the result on the top of the stack. The compiler then generates code to allocate a stack slot for `double` and store the result there.

As with all opcodes, the arguments are arranged on the stack with the leftmost argument on the top, while the return values are assumed to be laid out such that the rightmost variable is at the top of the stack.

Since `verbatim` can be used to generate arbitrary opcodes or even opcodes unknown to the Solidity compiler, care has to be taken when using `verbatim` together with the optimizer. Even when the optimizer is switched off, the code generator has to determine the stack layout, which means that e.g. using `verbatim` to modify the stack height can lead to undefined behavior.

The following is a non-exhaustive list of restrictions on verbatim bytecode that are not checked by the compiler. Violations of these restrictions can result in undefined behavior.

- Control-flow should not jump into or out of verbatim blocks, but it can jump within the same verbatim block. In particular, reverting or returning from the block is *not* allowed.
- Stack contents apart from the input and output parameters should not be accessed.
- The stack height difference should be exactly `m - n` (output slots minus input slots).
- Verbatim bytecode cannot make any assumptions about the surrounding bytecode. All required parameters have to be passed in as stack variables.

The optimizer does not analyze verbatim bytecode and always assumes that it modifies all aspects of state and thus can only do very few optimizations across `verbatim` function calls.

The optimizer treats verbatim bytecode as an opaque block of code. It will not split it but might move, duplicate or combine it with identical verbatim bytecode blocks. If a verbatim bytecode block is unreachable by the control-flow, it can be removed.

Warning

During discussions about whether or not EVM improvements might break existing smart contracts, features inside `verbatim` cannot receive the same consideration as those used by the Solidity compiler itself.

> **Note**
>
> To avoid confusion, all identifiers starting with the string `verbatim` are reserved and cannot be used for user-defined identifiers.

3.32.6 Specification of Yul Object

Yul objects are used to group named code and data sections. The functions `datasize`, `dataoffset` and `datacopy` can be used to access these sections from within code. Hex strings can be used to specify data in hex encoding, regular strings in native encoding. For code, `datacopy` will access its assembled binary representation.

```
Object = 'object' StringLiteral '{' Code ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral ( HexLiteral | StringLiteral )
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
```

Above, `Block` refers to `Block` in the Yul code grammar explained in the previous chapter.

> **Note**
>
> An object with a name that ends in `_deployed` is treated as deployed code by the Yul optimizer. The only consequence of this is a different gas cost heuristic in the optimizer.

> **Note**
>
> Data objects or sub-objects whose names contain a `.` can be defined but it is not possible to access them through `datasize`, `dataoffset` or `datacopy` because `.` is used as a separator to access objects inside another object.

> **Note**
>
> The data object called `".metadata"` has a special meaning: It cannot be accessed from code and is always appended to the very end of the bytecode, regardless of its position in the object.
>
> Other data objects with special significance might be added in the future, but their names will always start with a `.`.

An example Yul Object is shown below:

```
// A contract consists of a single object with sub-objects representing
// the code to be deployed or other contracts it can create.
// The single "code" node is the executable code of the object.
// Every (other) named object or data section is serialized and
// made accessible to the special built-in functions datacopy / dataoffset /␣
↪datasize
// The current object, sub-objects and data items inside the current object
// are in scope.
object "Contract1" {
    // This is the constructor code of the contract.
    code {
```

(continues on next page)

(continued from previous page)

```
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            // Note that Solidity generated IR code reserves memory offset ``0x60``␣
↪as well, but a pure Yul object is free to use memory as it chooses.
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // first create "Contract2"
        let size := datasize("Contract2")
        let offset := allocate(size)
        // This will turn into codecopy for EVM
        datacopy(offset, dataoffset("Contract2"), size)
        // constructor parameter is a single number 0x1234
        mstore(add(offset, size), 0x1234)
        pop(create(0, offset, add(size, 32)))

        // now return the runtime object (the currently
        // executing code is the constructor code)
        size := datasize("Contract1_deployed")
        offset := allocate(size)
        // This will turn into a codecopy for EVM
        datacopy(offset, dataoffset("Contract1_deployed"), size)
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "Contract1_deployed" {
        code {
            function allocate(size) -> ptr {
                ptr := mload(0x40)
                // Note that Solidity generated IR code reserves memory offset␣
↪``0x60`` as well, but a pure Yul object is free to use memory as it chooses.
                if iszero(ptr) { ptr := 0x60 }
                mstore(0x40, add(ptr, size))
            }

            // runtime code

            mstore(0, "Hello, World!")
            return(0, 0x20)
        }
    }

    // Embedded object. Use case is that the outside is a factory contract,
    // and Contract2 is the code to be created by the factory
    object "Contract2" {
        code {
            // code here ...
        }
```

(continues on next page)

(continued from previous page)

```
        object "Contract2_deployed" {
            code {
                // code here ...
            }
        }

        data "Table1" hex"4123"
    }
}
```

3.32.7 Yul Optimizer

The Yul optimizer operates on Yul code and uses the same language for input, output and intermediate states. This allows for easy debugging and verification of the optimizer.

Please refer to the general *optimizer documentation* for more details about the different optimization stages and how to use the optimizer.

If you want to use Solidity in stand-alone Yul mode, you activate the optimizer using `--optimize` and optionally specify the *expected number of contract executions* with `--optimize-runs`:

```
solc --strict-assembly --optimize --optimize-runs 200
```

In Solidity mode, the Yul optimizer is activated together with the regular optimizer.

Optimization Step Sequence

Detailed information regarding the optimization sequence as well as a list of abbreviations is available in the *optimizer docs*.

3.32.8 Complete ERC20 Example

```
object "Token" {
    code {
        // Store the creator in slot zero.
        sstore(0, caller())

        // Deploy the contract
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // Protection against sending Ether
            require(iszero(callvalue()))

            // Dispatcher
            switch selector()
            case 0x70a08231 /* "balanceOf(address)" */ {
                returnUint(balanceOf(decodeAsAddress(0)))
            }
            case 0x18160ddd /* "totalSupply()" */ {
                returnUint(totalSupply())
```

(continues on next page)

(continued from previous page)

```
            }
            case 0xa9059cbb /* "transfer(address,uint256)" */ {
                transfer(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
                transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
                returnTrue()
            }
            case 0x095ea7b3 /* "approve(address,uint256)" */ {
                approve(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0xdd62ed3e /* "allowance(address,address)" */ {
                returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
            }
            case 0x40c10f19 /* "mint(address,uint256)" */ {
                mint(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            default {
                revert(0, 0)
            }

            function mint(account, amount) {
                require(calledByOwner())

                mintTokens(amount)
                addToBalance(account, amount)
                emitTransfer(0, account, amount)
            }
            function transfer(to, amount) {
                executeTransfer(caller(), to, amount)
            }
            function approve(spender, amount) {
                revertIfZeroAddress(spender)
                setAllowance(caller(), spender, amount)
                emitApproval(caller(), spender, amount)
            }
            function transferFrom(from, to, amount) {
                decreaseAllowanceBy(from, caller(), amount)
                executeTransfer(from, to, amount)
            }

            function executeTransfer(from, to, amount) {
                revertIfZeroAddress(to)
                deductFromBalance(from, amount)
                addToBalance(to, amount)
                emitTransfer(from, to, amount)
            }
```

(continues on next page)

(continued from previous page)

```
            /* ---------- calldata decoding functions ----------- */
            function selector() -> s {
                s := div(calldataload(0),␣
↪0x100000000000000000000000000000000000000000000000000000000)
            }

            function decodeAsAddress(offset) -> v {
                v := decodeAsUint(offset)
                if iszero(iszero(and(v,␣
↪not(0xffffffffffffffffffffffffffffffffffffffff)))) {
                    revert(0, 0)
                }
            }
            function decodeAsUint(offset) -> v {
                let pos := add(4, mul(offset, 0x20))
                if lt(calldatasize(), add(pos, 0x20)) {
                    revert(0, 0)
                }
                v := calldataload(pos)
            }
            /* ---------- calldata encoding functions ---------- */
            function returnUint(v) {
                mstore(0, v)
                return(0, 0x20)
            }
            function returnTrue() {
                returnUint(1)
            }

            /* -------- events ---------- */
            function emitTransfer(from, to, amount) {
                let signatureHash :=␣
↪0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
                emitEvent(signatureHash, from, to, amount)
            }
            function emitApproval(from, spender, amount) {
                let signatureHash :=␣
↪0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
                emitEvent(signatureHash, from, spender, amount)
            }
            function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
                mstore(0, nonIndexed)
                log3(0, 0x20, signatureHash, indexed1, indexed2)
            }

            /* -------- storage layout ---------- */
            function ownerPos() -> p { p := 0 }
            function totalSupplyPos() -> p { p := 1 }
            function accountToStorageOffset(account) -> offset {
                offset := add(0x1000, account)
            }
            function allowanceStorageOffset(account, spender) -> offset {
```

(continues on next page)

(continued from previous page)

```
                offset := accountToStorageOffset(account)
                mstore(0, offset)
                mstore(0x20, spender)
                offset := keccak256(0, 0x40)
            }

            /* -------- storage access ---------- */
            function owner() -> o {
                o := sload(ownerPos())
            }
            function totalSupply() -> supply {
                supply := sload(totalSupplyPos())
            }
            function mintTokens(amount) {
                sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
            }
            function balanceOf(account) -> bal {
                bal := sload(accountToStorageOffset(account))
            }
            function addToBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                sstore(offset, safeAdd(sload(offset), amount))
            }
            function deductFromBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                let bal := sload(offset)
                require(lte(amount, bal))
                sstore(offset, sub(bal, amount))
            }
            function allowance(account, spender) -> amount {
                amount := sload(allowanceStorageOffset(account, spender))
            }
            function setAllowance(account, spender, amount) {
                sstore(allowanceStorageOffset(account, spender), amount)
            }
            function decreaseAllowanceBy(account, spender, amount) {
                let offset := allowanceStorageOffset(account, spender)
                let currentAllowance := sload(offset)
                require(lte(amount, currentAllowance))
                sstore(offset, sub(currentAllowance, amount))
            }

            /* ---------- utility functions ---------- */
            function lte(a, b) -> r {
                r := iszero(gt(a, b))
            }
            function gte(a, b) -> r {
                r := iszero(lt(a, b))
            }
            function safeAdd(a, b) -> r {
                r := add(a, b)
                if or(lt(r, a), lt(r, b)) { revert(0, 0) }
```

(continues on next page)

(continued from previous page)

```
            }
            function calledByOwner() -> cbo {
                cbo := eq(owner(), caller())
            }
            function revertIfZeroAddress(addr) {
                require(addr)
            }
            function require(condition) {
                if iszero(condition) { revert(0, 0) }
            }
        }
    }
}
```

3.33 Import Path Resolution

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. Paths used in imports must work the same way everywhere while the command-line interface must be able to work with platform-specific paths to provide good user experience. This section aims to explain in detail how Solidity reconciles these requirements.

3.33.1 Virtual Filesystem

The compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. When you use the *import statement*, you specify an *import path* that references a source unit name.

Import Callback

The VFS is initially populated only with files the compiler has received as input. Additional files can be loaded during compilation using an *import callback*, which is different depending on the type of compiler you use (see below). If the compiler does not find any source unit name matching the import path in the VFS, it invokes the callback, which is responsible for obtaining the source code to be placed under that name. An import callback is free to interpret source unit names in an arbitrary way, not just as paths. If there is no callback available when one is needed or if it fails to locate the source code, compilation fails.

By default, the command-line compiler provides the *Host Filesystem Loader* - a rudimentary callback that interprets a source unit name as a path in the local filesystem. This callback can be disabled using the `--no-import-callback` command-line option. The JavaScript interface does not provide any by default, but one can be provided by the user. This mechanism can be used to obtain source code from locations other than the local filesystem (which may not even be accessible, e.g. when the compiler is running in a browser). For example the Remix IDE provides a versatile callback that lets you import files from HTTP, IPFS and Swarm URLs or refer directly to packages in NPM registry.

Note

Host Filesystem Loader's file lookup is platform-dependent. For example backslashes in a source unit name can be interpreted as directory separators or not and the lookup can be case-sensitive or not, depending on the underlying platform.

For portability it is recommended to avoid using import paths that will work correctly only with a specific import callback or only on one platform. For example you should always use forward slashes since they work as path separators also on platforms that support backslashes.

Initial Content of the Virtual Filesystem

The initial content of the VFS depends on how you invoke the compiler:

1. **solc / command-line interface**

   When you compile a file using the command-line interface of the compiler, you provide one or more paths to files containing Solidity code:

   ```
   solc contract.sol /usr/local/dapp-bin/token.sol
   ```

   The source unit name of a file loaded this way is constructed by converting its path to a canonical form and, if possible, making it relative to either the base path or one of the include paths. See *CLI Path Normalization and Stripping* for a detailed description of this process.

2. **Standard JSON**

   When using the *Standard JSON* API (via either the JavaScript interface or the `--standard-json` command-line option) you provide input in JSON format, containing, among other things, the content of all your source files:

   ```
   {
       "language": "Solidity",
       "sources": {
           "contract.sol": {
               "content": "import \"./util.sol\";\ncontract C {}"
           },
           "util.sol": {
               "content": "library Util {}"
           },
           "/usr/local/dapp-bin/token.sol": {
               "content": "contract Token {}"
           }
       },
       "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
   }
   ```

   The `sources` dictionary becomes the initial content of the virtual filesystem and its keys are used as source unit names.

3. **Standard JSON (via import callback)**

   With Standard JSON it is also possible to tell the compiler to use the import callback to obtain the source code:

   ```
   {
       "language": "Solidity",
       "sources": {
           "/usr/local/dapp-bin/token.sol": {
               "urls": [
                   "/projects/mytoken.sol",
                   "https://example.com/projects/mytoken.sol"
               ]
           }
       },
       "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
   }
   ```

   If an import callback is available, the compiler will give it the strings specified in `urls` one by one, until one is loaded successfully or the end of the list is reached.

The source unit names are determined the same way as when using `content` - they are keys of the `sources` dictionary and the content of `urls` does not affect them in any way.

4. **Standard input**

   On the command-line it is also possible to provide the source by sending it to compiler's standard input:

   ```
   echo 'import "./util.sol"; contract C {}' | solc -
   ```

   `-` used as one of the arguments instructs the compiler to place the content of the standard input in the virtual filesystem under a special source unit name: `<stdin>`.

Once the VFS is initialized, additional files can still be added to it only through the import callback.

3.33.2 Imports

The import statement specifies an *import path*. Based on how the import path is specified, we can divide imports into two categories:

- *Direct imports*, where you specify the full source unit name directly.
- *Relative imports*, where you specify a path starting with `./` or `../` to be combined with the source unit name of the importing file.

Listing 1: contracts/contract.sol

```
import "./math/math.sol";
import "contracts/tokens/token.sol";
```

In the above `./math/math.sol` and `contracts/tokens/token.sol` are import paths while the source unit names they translate to are `contracts/math/math.sol` and `contracts/tokens/token.sol` respectively.

Direct Imports

An import that does not start with `./` or `../` is a *direct import*.

```
import "/project/lib/util.sol";         // source unit name: /project/lib/util.sol
import "lib/util.sol";                  // source unit name: lib/util.sol
import "@openzeppelin/address.sol";     // source unit name: @openzeppelin/address.sol
import "https://example.com/token.sol"; // source unit name: https://example.com/token.
↪sol
```

After applying any *import remappings* the import path simply becomes the source unit name.

Note

A source unit name is just an identifier and even if its value happens to look like a path, it is not subject to the normalization rules you would typically expect in a shell. Any `/./` or `/../` segments or sequences of multiple slashes remain a part of it. When the source is provided via Standard JSON interface it is entirely possible to associate different content with source unit names that would refer to the same file on disk.

When the source is not available in the virtual filesystem, the compiler passes the source unit name to the import callback. The Host Filesystem Loader will attempt to use it as a path and look up the file on disk. At this point the platform-specific normalization rules kick in and names that were considered different in the VFS may actually result in the same file being loaded. For example `/project/lib/math.sol` and `/project/lib/../lib///math.sol` are considered completely different in the VFS even though they refer to the same file on disk.

> **Note**
>
> Even if an import callback ends up loading source code for two different source unit names from the same file on disk, the compiler will still see them as separate source units. It is the source unit name that matters, not the physical location of the code.

Relative Imports

An import starting with `./` or `../` is a *relative import*. Such imports specify a path relative to the source unit name of the importing source unit:

Listing 2: /project/lib/math.sol

```
import "./util.sol" as util;    // source unit name: /project/lib/util.sol
import "../token.sol" as token; // source unit name: /project/token.sol
```

Listing 3: lib/math.sol

```
import "./util.sol" as util;    // source unit name: lib/util.sol
import "../token.sol" as token; // source unit name: token.sol
```

> **Note**
>
> Relative imports **always** start with `./` or `../` so `import "util.sol"`, unlike `import "./util.sol"`, is a direct import. While both paths would be considered relative in the host filesystem, `util.sol` is actually absolute in the VFS.

Let us define a *path segment* as any non-empty part of the path that does not contain a separator and is bounded by two path separators. A separator is a forward slash or the beginning/end of the string. For example in `./abc/..//` there are three path segments: `.`, `abc` and `..`.

The compiler resolves the import into a source unit name based on the import path, in the following way:

1. We start with the source unit name of the importing source unit.
2. The last path segment with preceding slashes is removed from the resolved name.
3. Then, for every segment in the import path, starting from the leftmost one:
   - If the segment is `.`, it is skipped.
   - If the segment is `..`, the last path segment with preceding slashes is removed from the resolved name.
   - Otherwise, the segment (preceded by a single slash if the resolved name is not empty), is appended to the resolved name.

The removal of the last path segment with preceding slashes is understood to work as follows:

1. Everything past the last slash is removed (i.e. `a/b//c.sol` becomes `a/b//`).
2. All trailing slashes are removed (i.e. `a/b//` becomes `a/b`).

Note that the process normalizes the part of the resolved source unit name that comes from the import path according to the usual rules for UNIX paths, i.e. all `.` and `..` are removed and multiple slashes are squashed into a single one. On the other hand, the part that comes from the source unit name of the importing module remains unnormalized. This ensures that the `protocol://` part does not turn into `protocol:/` if the importing file is identified with a URL.

If your import paths are already normalized, you can expect the above algorithm to produce very intuitive results. Here are some examples of what you can expect if they are not:

Listing 4: lib/src/../contract.sol

```
import "./util/./util.sol";         // source unit name: lib/src/../util/util.sol
import "./util//util.sol";          // source unit name: lib/src/../util/util.sol
import "../util/../array/util.sol"; // source unit name: lib/src/array/util.sol
import "../../../../util.sol";      // source unit name: util.sol
import "../../../../../util.sol";   // source unit name: util.sol
```

Note

The use of relative imports containing leading `..` segments is not recommended. The same effect can be achieved in a more reliable way by using direct imports with *base path and include paths*.

3.33.3 Base Path and Include Paths

The base path and include paths represent directories that the Host Filesystem Loader will load files from. When a source unit name is passed to the loader, it prepends the base path to it and performs a filesystem lookup. If the lookup does not succeed, the same is done with all directories on the include path list.

It is recommended to set the base path to the root directory of your project and use include paths to specify additional locations that may contain libraries your project depends on. This lets you import from these libraries in a uniform way, no matter where they are located in the filesystem relative to your project. For example, if you use npm to install packages and your contract imports `@openzeppelin/contracts/utils/Strings.sol`, you can use these options to tell the compiler that the library can be found in one of the npm package directories:

```
solc contract.sol \
    --base-path . \
    --include-path node_modules/ \
    --include-path /usr/local/lib/node_modules/
```

Your contract will compile (with the same exact metadata) no matter whether you install the library in the local or global package directory or even directly under your project root.

By default the base path is empty, which leaves the source unit name unchanged. When the source unit name is a relative path, this results in the file being looked up in the directory the compiler has been invoked from. It is also the only value that results in absolute paths in source unit names being actually interpreted as absolute paths on disk. If the base path itself is relative, it is interpreted as relative to the current working directory of the compiler.

Note

Include paths cannot have empty values and must be used together with a non-empty base path.

Note

Include paths and base path can overlap as long as it does not make import resolution ambiguous. For example, you can specify a directory inside base path as an include directory or have an include directory that is a subdirectory of another include directory. The compiler will only issue an error if the source unit name passed to the Host

Filesystem Loader represents an existing path when combined with multiple include paths or an include path and base path.

CLI Path Normalization and Stripping

On the command-line the compiler behaves just as you would expect from any other program: it accepts paths in a format native to the platform and relative paths are relative to the current working directory. The source unit names assigned to files whose paths are specified on the command-line, however, should not change just because the project is being compiled on a different platform or because the compiler happens to have been invoked from a different directory. To achieve this, paths to source files coming from the command-line must be converted to a canonical form, and, if possible, made relative to the base path or one of the include paths.

The normalization rules are as follows:

- If a path is relative, it is made absolute by prepending the current working directory to it.
- Internal `.` and `..` segments are collapsed.
- Platform-specific path separators are replaced with forward slashes.
- Sequences of multiple consecutive path separators are squashed into a single separator (unless they are the leading slashes of an UNC path).
- If the path includes a root name (e.g. a drive letter on Windows) and the root is the same as the root of the current working directory, the root is replaced with `/`.
- Symbolic links in the path are **not** resolved.
  - The only exception is the path to the current working directory prepended to relative paths in the process of making them absolute. On some platforms the working directory is reported always with symbolic links resolved so for consistency the compiler resolves them everywhere.
- The original case of the path is preserved even if the filesystem is case-insensitive but case-preserving and the actual case on disk is different.

Note

There are situations where paths cannot be made platform-independent. For example on Windows the compiler can avoid using drive letters by referring to the root directory of the current drive as `/` but drive letters are still necessary for paths leading to other drives. You can avoid such situations by ensuring that all the files are available within a single directory tree on the same drive.

After normalization the compiler attempts to make the source file path relative. It tries the base path first and then the include paths in the order they were given. If the base path is empty or not specified, it is treated as if it was equal to the path to the current working directory (with all symbolic links resolved). The result is accepted only if the normalized directory path is the exact prefix of the normalized file path. Otherwise the file path remains absolute. This makes the conversion unambiguous and ensures that the relative path does not start with `../`. The resulting file path becomes the source unit name.

Note

The relative path produced by stripping must remain unique within the base path and include paths. For example the compiler will issue an error for the following command if both `/project/contract.sol` and `/lib/contract.sol` exist:

```
solc /project/contract.sol --base-path /project --include-path /lib
```

Note

Prior to version 0.8.8, CLI path stripping was not performed and the only normalization applied was the conversion of path separators. When working with older versions of the compiler it is recommended to invoke the compiler from the base path and to only use relative paths on the command-line.

3.33.4 Allowed Paths

As a security measure, the Host Filesystem Loader will refuse to load files from outside of a few locations that are considered safe by default:

- Outside of Standard JSON mode:
  - The directories containing input files listed on the command-line.
  - The directories used as *remapping* targets. If the target is not a directory (i.e does not end with `/`, `/.` or `/..`) the directory containing the target is used instead.
  - Base path and include paths.
- In Standard JSON mode:
  - Base path and include paths.

Additional directories can be whitelisted using the `--allow-paths` option. The option accepts a comma-separated list of paths:

```
cd /home/user/project/
solc token/contract.sol \
    lib/util.sol=libs/util.sol \
    --base-path=token/ \
    --include-path=/lib/ \
    --allow-paths=../utils/,/tmp/libraries
```

When the compiler is invoked with the command shown above, the Host Filesystem Loader will allow importing files from the following directories:

- `/home/user/project/token/` (because `token/` contains the input file and also because it is the base path),
- `/lib/` (because `/lib/` is one of the include paths),
- `/home/user/project/libs/` (because `libs/` is a directory containing a remapping target),
- `/home/user/utils/` (because of `../utils/` passed to `--allow-paths`),
- `/tmp/libraries/` (because of `/tmp/libraries` passed to `--allow-paths`),

Note

The working directory of the compiler is one of the paths allowed by default only if it happens to be the base path (or the base path is not specified or has an empty value).

Note

The compiler does not check if allowed paths actually exist and whether they are directories. Non-existent or empty paths are simply ignored. If an allowed path matches a file rather than a directory, the file is considered whitelisted,

too.

Note

Allowed paths are case-sensitive even if the filesystem is not. The case must exactly match the one used in your imports. For example `--allow-paths tokens` will not match `import "Tokens/IERC20.sol"`.

Warning

Files and directories only reachable through symbolic links from allowed directories are not automatically whitelisted. For example if `token/contract.sol` in the example above was actually a symlink pointing at `/etc/passwd` the compiler would refuse to load it unless `/etc/` was one of the allowed paths too.

3.33.5 Import Remapping

Import remapping allows you to redirect imports to a different location in the virtual filesystem. The mechanism works by changing the translation between import paths and source unit names. For example you can set up a remapping so that any import from the virtual directory `github.com/ethereum/dapp-bin/library/` would be seen as an import from `dapp-bin/library/` instead.

You can limit the scope of a remapping by specifying a *context*. This allows creating remappings that apply only to imports located in a specific library or a specific file. Without a context a remapping is applied to every matching import in all the files in the virtual filesystem.

Import remappings have the form of `context:prefix=target`:

- `context` must match the beginning of the source unit name of the file containing the import.
- `prefix` must match the beginning of the source unit name resulting from the import.
- `target` is the value the prefix is replaced with.

For example, if you clone https://github.com/ethereum/dapp-bin/ locally to `/project/dapp-bin` and run the compiler with:

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // source unit name: dapp-bin/
↪library/math.sol
```

The compiler will look for the file in the VFS under `dapp-bin/library/math.sol`. If the file is not available there, the source unit name will be passed to the Host Filesystem Loader, which will then look in `/project/dapp-bin/library/math.sol`.

Warning

Information about remappings is stored in contract metadata. Since the binary produced by the compiler has a hash of the metadata embedded in it, any modification to the remappings will result in different bytecode.

For this reason you should be careful not to include any local information in remapping targets. For example if your library is located in `/home/user/packages/mymath/math.sol`, a remapping like `@math/=/home/user/packages/mymath/` would result in your home directory being included in the metadata. To be able to reproduce the same bytecode with such a remapping on a different machine, you would need to recreate parts of your local directory structure in the VFS and (if you rely on Host Filesystem Loader) also in the host filesystem.

To avoid having your local directory structure embedded in the metadata, it is recommended to designate the directories containing libraries as *include paths* instead. For example, in the example above `--include-path /home/user/packages/` would let you use imports starting with `mymath/`. Unlike remapping, the option on its own will not make `mymath` appear as `@math` but this can be achieved by creating a symbolic link or renaming the package subdirectory.

As a more complex example, suppose you rely on a module that uses an old version of dapp-bin that you checked out to `/project/dapp-bin_old`, then you can run:

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \
     --base-path /project \
     source.sol
```

This means that all imports in `module2` point to the old version but imports in `module1` point to the new version.

Here are the detailed rules governing the behavior of remappings:

1. **Remappings only affect the translation between import paths and source unit names.**

   Source unit names added to the VFS in any other way cannot be remapped. For example the paths you specify on the command-line and the ones in `sources.urls` in Standard JSON are not affected.

   ```
   solc /project/=/contracts/ /project/contract.sol # source unit name: /project/
   ↪contract.sol
   ```

   In the example above the compiler will load the source code from `/project/contract.sol` and place it under that exact source unit name in the VFS, not under `/contract/contract.sol`.

2. **Context and prefix must match source unit names, not import paths.**

   - This means that you cannot remap `./` or `../` directly since they are replaced during the translation to source unit name but you can remap the part of the name they are replaced with:

     ```
     solc ./=a/ /project/=b/ /project/contract.sol # source unit name: /project/
     ↪contract.sol
     ```

     Listing 5: /project/contract.sol

     ```
     import "./util.sol" as util; // source unit name: b/util.sol
     ```

   - You cannot remap base path or any other part of the path that is only added internally by an import callback:

     ```
     solc /project/=/contracts/ /project/contract.sol --base-path /project # source␣
     ↪unit name: contract.sol
     ```

Listing 6: /project/contract.sol

```
import "util.sol" as util; // source unit name: util.sol
```

3. **Target is inserted directly into the source unit name and does not necessarily have to be a valid path.**
   - It can be anything as long as the import callback can handle it. In case of the Host Filesystem Loader this includes also relative paths. When using the JavaScript interface you can even use URLs and abstract identifiers if your callback can handle them.
   - Remapping happens after relative imports have already been resolved into source unit names. This means that targets starting with `./` and `../` have no special meaning and are relative to the base path rather than to the location of the source file.
   - Remapping targets are not normalized so `@root/=./a/b//` will remap `@root/contract.sol` to `./a/b//contract.sol` and not `a/b/contract.sol`.
   - If the target does not end with a slash, the compiler will not add one automatically:

```
solc /project/=/contracts /project/contract.sol # source unit name: /project/
↪contract.sol
```

Listing 7: /project/contract.sol

```
import "/project/util.sol" as util; // source unit name: /contractsutil.sol
```

4. **Context and prefix are patterns and matches must be exact.**
   - `a//b=c` will not match `a/b`.
   - source unit names are not normalized so `a/b=c` will not match `a//b` either.
   - Parts of file and directory names can match as well. `/newProject/con:/new=old` will match `/newProject/contract.sol` and remap it to `oldProject/contract.sol`.
5. **At most one remapping is applied to a single import.**
   - If multiple remappings match the same source unit name, the one with the longest matching context is chosen.
   - If contexts are identical, the one with the longest matching prefix is chosen.
   - If contexts and prefixes are identical, the one specified last wins.
   - Remappings do not work on other remappings. For example `a=b b=c c=d` will not result in `a` being remapped to `d`.
6. **Prefix cannot be empty but context and target are optional.**
   - If `target` is the empty string, `prefix` is simply removed from import paths.
   - Empty `context` means that the remapping applies to all imports in all source units.

3.33.6 Using URLs in imports

Most URL prefixes such as `https://` or `data://` have no special meaning in import paths. The only exception is `file://` which is stripped from source unit names by the Host Filesystem Loader.

When compiling locally you can use import remapping to replace the protocol and domain part with a local path:

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

Note the leading `:`, which is necessary when the remapping context is empty. Otherwise the `https:` part would be interpreted by the compiler as the context.

3.34 Style Guide

3.34.1 Introduction

This guide is intended to provide coding conventions for writing Solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

The structure and many of the recommendations within this style guide were taken from Python's pep8 style guide.

The goal of this guide is *not* to be the right way or the best way to write Solidity code. The goal of this guide is *consistency*. A quote from Python's pep8 captures this concept well.

> **Note**
>
> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.
>
> But most importantly: **know when to be inconsistent** – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And do not hesitate to ask!

3.34.2 Code Layout

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

Blank Lines

Surround top level declarations in Solidity source with two blank lines.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}


contract B {
    // ...
}
```

(continues on next page)

(continued from previous page)

```
contract C {
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() virtual pure public;
```

(continues on next page)

(continued from previous page)

```
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

Maximum Line Length

Maximum suggested line length is 120 characters.

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.
2. One, and only one, indent should be used.
3. Each argument should fall on its own line.
4. The terminating element, `);`, should be placed on the final line by itself.

Function Calls

Yes:

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No:

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
```

(continues on next page)

(continued from previous page)

```
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Assignment Statements

Yes:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(argument1,
                                                                 argument2,
                                                                 argument3,
                                                                 argument4);
```

Event Definitions and Event Emitters

Yes:

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

emit LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

No:

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);
```

(continues on next page)

(continued from previous page)

```
emit LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";

contract A {
    // ...
}


contract B is Owned {
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}


import "./Owned.sol";


contract B is Owned {
    // ...
}
```

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

- constructor

- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the `view` and `pure` functions last.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // External functions
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {

    // External functions
```

(continues on next page)

(continued from previous page)

```
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() {
        // ...
    }

    // Internal functions
    // ...
}
```

Whitespace in Expressions

Avoid extraneous whitespace in the following situations:

Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.

Yes:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception:

```
function singleLine() public { spam(); }
```

Immediately before a comma, semicolon:

Yes:

```
function spam(uint i, Coin coin) public;
```

No:

```
function spam(uint i , Coin coin) public ;
```

More than one space around an assignment or other operator to align with another:

Yes:

```
x = 1;
y = 2;
longVariable = 3;
```

No:

```
x            = 1;
y            = 2;
longVariable = 3;
```

Do not include a whitespace in the receive and fallback functions:

Yes:

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

No:

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

Control Structures

The braces denoting the body of a contract, library, functions and structs should:

- open on the same line as the declaration
- close on their own line at the same indentation level as the beginning of the declaration.
- The opening brace should be preceded by a single space.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

The same recommendations apply to the control structures `if`, `else`, `while`, and `for`.

Additionally there should be a single space between the control structures `if`, `while`, and `for` and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Yes:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

For control structures whose body contains a single statement, omitting the braces is ok *if* the statement is contained on a single line.

Yes:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

For `if` blocks which have an `else` or `else if` clause, the `else` should be placed on the same line as the `if`'s closing brace. This is an exception compared to the rules of other block-like structures.

Yes:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function declaration.

The opening brace should be preceded by a single space.

Yes:

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
```

(continues on next page)

(continued from previous page)

```
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

The modifier order for a function should be:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

Yes:

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

No:

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function increment(uint x) onlyOwner public pure returns (uint) {
    return x + 1;
}
```

For long function declarations, it is recommended to drop each argument onto its own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the same indentation level as the function declaration.

Yes:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
```

(continues on next page)

(continued from previous page)

```
{
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to its own line.

Yes:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z
)
    public
    onlyOwner
    priced
    returns (address)
{
```

(continues on next page)

(continued from previous page)

```
    doSomething();
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyOwner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyOwner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address) {
    doSomething();
}
```

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the *Maximum Line Length* section.

Yes:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

No:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
        x = param5;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4) {
        x = param5;
    }
}


contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
            x = param5;
        }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible:

```
function shortFunction() public { doSomething(); }
```

These guidelines for function declarations are intended to improve readability. Authors should use their best judgment as this guide does not try to cover all possible permutations for function declarations.

Mappings

In variable declarations, do not separate the keyword `mapping` from its type by a space. Do not separate any nested `mapping` keyword from its type by whitespace.

Yes:

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

No:

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

Variable Declarations

Declarations of array variables should not have a space between the type and the brackets.

Yes:

```
uint[] x;
```

No:

```
uint [] x;
```

Other Recommendations

- Strings should be quoted with double-quotes instead of single-quotes.

Yes:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Surround operators with a single space on either side.

Yes:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statements. You should always use the same amount of whitespace on either side of an operator:

Yes:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

3.34.3 Order of Layout

Contract elements should be laid out in the following order:

1. Pragma statements
2. Import statements
3. Events
4. Errors
5. Interfaces
6. Libraries
7. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions

Note

It might be clearer to declare types close to their use in events or state variables.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.4 <0.9.0;

abstract contract Math {
    error DivideByZero();
    function divide(int256 numerator, int256 denominator) public virtual returns␣
↪(uint256);
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.4 <0.9.0;

abstract contract Math {
    function divide(int256 numerator, int256 denominator) public virtual returns␣
↪(uint256);
    error DivideByZero();
}
```

3.34.4 Naming Conventions

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant *meta* information that would otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Lastly, consistency within a codebase should always supersede any conventions outlined in this document.

Naming Styles

To avoid confusion, the following names will be used to refer to different naming styles.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords)
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)

Note

When using initialisms in CapWords, capitalize all the letters of the initialisms. Thus HTTPServerError is better than HttpServerError. When using initialisms in mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus xmlHTTPRequest is better than XMLHTTPRequest.

Names to Avoid

- `l` - Lowercase letter el
- `O` - Uppercase letter oh
- `I` - Uppercase letter eye

Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.

Contract and Library Names

- Contracts and libraries should be named using the CapWords style. Examples: `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
- Contract and library names should also match their filenames.
- If a contract file includes multiple contracts and/or libraries, then the filename should match the *core contract*. This is not recommended however if it can be avoided.

As shown in the example below, if the contract name is `Congress` and the library name is `Owned`, then their associated filenames should be `Congress.sol` and `Owned.sol`.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
```

(continues on next page)

(continued from previous page)

```
import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;


import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

Struct Names

Structs should be named using the CapWords style. Examples: `MyCoin`, `Position`, `PositionXY`.

Event Names

Events should be named using the CapWords style. Examples: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

Function Names

Functions should use mixedCase. Examples: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Function Argument Names

Function arguments should use mixedCase. Examples: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

Local and State Variable Names

Use mixedCase. Examples: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constants

Constants should be named with all capital letters with underscores separating words. Examples: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Modifier Names

Use mixedCase. Examples: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Avoiding Naming Collisions

- `singleTrailingUnderscore_`

This convention is suggested when the desired name collides with that of an existing state variable, function, built-in or otherwise reserved name.

Underscore Prefix for Non-external Functions and Variables

- `_singleLeadingUnderscore`

This convention is suggested for non-external functions and state variables (`private` or `internal`). State variables without a specified visibility are `internal` by default.

When designing a smart contract, the public-facing API (functions that can be called by any account) is an important consideration. Leading underscores allow you to immediately recognize the intent of such functions, but more importantly, if you change a function from non-external to external (including `public`) and rename it accordingly, this forces you to review every call site while renaming. This can be an important manual check against unintended external functions and a common source of security vulnerabilities (avoid find-replace-all tooling for this change).

3.34.5 NatSpec

Solidity contracts can also contain NatSpec comments. They are written with a triple slash (`///`) or a double asterisk block (`/** ... */`) and they should be used directly above function declarations or statements.

For example, the contract from *a simple smart contract* with the comments added looks like the one below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
    uint storedData;

    /// Store `x`.
    /// @param x the new value to store
    /// @dev stores the number in the state variable `storedData`
    function set(uint x) public {
        storedData = x;
    }

    /// Return the stored value.
    /// @dev retrieves the value of the state variable `storedData`
    /// @return the stored value
    function get() public view returns (uint) {
        return storedData;
    }
}
```

It is recommended that Solidity contracts are fully annotated using *NatSpec* for all public interfaces (everything in the ABI).

Please see the section about *NatSpec* for a detailed explanation.

3.35 Common Patterns

3.35.1 Withdrawal from Contracts

The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending Ether, as a result of an effect, is a direct `transfer` call, this is not recommended as it introduces a potential security risk. You may read more about this on the *Security Considerations* page.

The following is an example of the withdrawal pattern in practice in a contract where the goal is to send the most of some compensation, e.g. Ether, to the contract in order to become the "richest", inspired by King of the Ether.

In the following contract, if you are no longer the richest, you receive the funds of the person who is now the richest.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping(address => uint) pendingWithdrawals;

    /// The amount of Ether sent was not higher than
    /// the currently highest amount.
    error NotEnoughEther();
```

(continues on next page)

(continued from previous page)

```
    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent reentrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success);
    }
}
```

This is as opposed to the more intuitive sending pattern:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    /// The amount of Ether sent was not higher than
    /// the currently highest amount.
    error NotEnoughEther();

    constructor() payable {
        richest = payable(msg.sender);
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        // This line can cause problems (explained below).
        (bool success, ) = richest.call{value: msg.value}("");
        require(success);
        richest = payable(msg.sender);
        mostSent = msg.value;
    }
}
```

Notice that, in this example, an attacker could trap the contract into an unusable state by causing `richest` to be the address of a contract that has a receive or fallback function which fails (e.g. by using `revert()` or by just consuming more than the 2300 gas stipend transferred to them). That way, whenever `transfer` is called to deliver funds to the

"poisoned" contract, it will fail and thus also `becomeRichest` will fail, with the contract being stuck forever.

In contrast, if you use the "withdraw" pattern from the first example, the attacker can only cause his or her own withdraw to fail and not the rest of the contract's workings.

3.35.2 Restricting Access

Restricting access is a common pattern for contracts. Note that you can never restrict any human or computer from reading the content of your transactions or your contract's state. You can make it a bit harder by using encryption, but if your contract is supposed to read the data, so will everyone else.

You can restrict read access to your contract's state by **other contracts**. That is actually the default unless you declare your state variables `public`.

Furthermore, you can restrict who can make modifications to your contract's state or call your contract's functions and this is what this section is about.

The use of **function modifiers** makes these restrictions highly readable.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
    // creating this contract.
    address public owner = msg.sender;
    uint public creationTime = block.timestamp;

    // Now follows a list of errors that
    // this contract can generate together
    // with a textual explanation in special
    // comments.

    /// Sender not authorized for this
    /// operation.
    error Unauthorized();

    /// Function called too early.
    error TooEarly();

    /// Not enough Ether sent with function call.
    error NotEnoughEther();

    // Modifiers can be used to change
    // the body of a function.
    // If this modifier is used, it will
    // prepend a check that only passes
    // if the function is called from
    // a certain address.
    modifier onlyBy(address account)
    {
        if (msg.sender != account)
            revert Unauthorized();
        // Do not forget the "_;"! It will
        // be replaced by the actual function
```

(continues on next page)

(continued from previous page)

```
        // body when the modifier is used.
        _;
    }

    /// Make `newOwner` the new owner of this
    /// contract.
    function changeOwner(address newOwner)
        public
        onlyBy(owner)
    {
        owner = newOwner;
    }

    modifier onlyAfter(uint time) {
        if (block.timestamp < time)
            revert TooEarly();
        _;
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // This modifier requires a certain
    // fee being associated with a function call.
    // If the caller sent too much, he or she is
    // refunded, but only after the function body.
    // This was dangerous before Solidity version 0.4.0,
    // where it was possible to skip the part after `_;`.
    modifier costs(uint amount) {
        if (msg.value < amount)
            revert NotEnoughEther();

        _;
        if (msg.value > amount) {
            (bool success, ) = payable(msg.sender).call{value: msg.value - amount}("");
            require(success);
        }
    }

    function forceOwnerChange(address newOwner)
        public
        payable
        costs(200 ether)
    {
```

(continues on next page)

(continued from previous page)

```
        owner = newOwner;
        // just some example condition
        if (uint160(owner) & 0 == 1)
            // This did not refund for Solidity
            // before version 0.4.0.
            return;
        // refund overpaid fees
    }
}
```

A more specialised way in which access to function calls can be restricted will be discussed in the next example.

3.35.3 State Machine

Contracts often act as a state machine, which means that they have certain **stages** in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models **interaction**). It is also common that some stages are automatically reached at a certain point in **time**.

An example for this is a blind auction contract which starts in the stage "accepting blinded bids", then transitions to "revealing bids" which is ended by "determine auction outcome".

Function modifiers can be used in this situation to model the states and guard against incorrect usage of the contract.

Example

In the following example, the modifier `atStage` ensures that the function can only be called at a certain stage.

Automatic timed transitions are handled by the modifier `timedTransitions`, which should be used for all functions.

> **Note**
>
> **Modifier Order Matters**. If atStage is combined with timedTransitions, make sure that you mention it after the latter, so that the new stage is taken into account.

Finally, the modifier `transitionNext` can be used to automatically go to the next stage when the function finishes.

> **Note**
>
> **Modifier May be Skipped**. This only applies to Solidity before version 0.4.0: Since modifiers are applied by simply replacing code and not by using a function call, the code in the transitionNext modifier can be skipped if the function itself uses return. If you want to do that, make sure to call nextStage manually from those functions. Starting with version 0.4.0, modifier code will run even if the function explicitly returns.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
```

(continues on next page)

(continued from previous page)

```
        AreWeDoneYet,
        Finished
    }
    /// Function cannot be called at this time.
    error FunctionInvalidAtThisStage();

    // This is the current stage.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = block.timestamp;

    modifier atStage(Stages stage_) {
        if (stage != stage_)
            revert FunctionInvalidAtThisStage();
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Perform timed transitions. Be sure to mention
    // this modifier first, otherwise the guards
    // will not take the new stage into account.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    block.timestamp >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                block.timestamp >= creationTime + 12 days)
            nextStage();
        // The other stages transition by transaction
        _;
    }

    // Order of the modifiers matters here!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }
```

(continues on next page)

(continued from previous page)

```
    // This modifier goes to the next stage
    // after the function is done.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

3.36 Resources

3.36.1 General Resources

- Ethereum.org Developers page
- Ethereum StackExchange
- Solidity website
- Solidity changelog
- Solidity codebase on GitHub
- Solidity language users chat
- Solidity compiler developers chat
- awesome-solidity
- Solidity by Example
- Solidity documentation community translations
- Solidity and Smart Contract Glossary

3.36.2 Integrated (Ethereum) Development Environments

- **Ape**
  A Python-based web3 development tool for compiling, testing, and interacting with smart contracts.
- **Brownie**
  A Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine. Note: As per the official docs, Brownie is no longer actively maintained. Future releases may come sporadically - or never at all. Check out Ape Framework (first in list) for all your python Ethereum development needs.
- **Dapp**
  Tool for building, testing and deploying smart contracts from the command-line.
- **Foundry**
  Fast, portable and modular toolkit for Ethereum application development written in Rust.
- **Hardhat**
  Ethereum development environment with local Ethereum network, debugging features and plugin ecosystem.
- **Remix**
  Browser-based IDE with integrated compiler and Solidity runtime environment without server-side components.
- **Truffle**
  Ethereum development framework. Note: Consensys announced the sunset of Truffle on September 21, 2023. Current users may check out the migration path and available product support here.

3.36.3 Editor Integrations

- Emacs
  - **Emacs Solidity**
    Plugin for the Emacs editor providing syntax highlighting and compilation error reporting.
- IntelliJ
  - **IntelliJ IDEA plugin**
    Solidity plugin for IntelliJ IDEA (and all other JetBrains IDEs).
- Sublime Text
  - **Package for SublimeText - Solidity language syntax**
    Solidity syntax highlighting for SublimeText editor.
- Vim
  - **Vim Solidity by Thesis**
    Syntax highlighting for Solidity in Vim.
  - **Vim Solidity by TovarishFin**
    Vim syntax file for Solidity.
  - **Vim Syntastic**
    Plugin for the Vim editor providing compile checking.
- Visual Studio Code (VS Code)
  - **Aderyn Visual Studio Code extension**
    Solidity Smart contract analyzer designed to help find vulnerabilities. It supports projects built with Hardhat, Foundry, or any custom framework.

- **Ethereum Remix Visual Studio Code extension**
  Ethereum Remix extension pack for VS Code Note: As per the official repository, this extension has been removed from the VSCODE marketplace and will be replaced by a dedicated stand-alone desktop application.
- **Solidity Visual Studio Code extension, by Juan Blanco**
  Solidity plugin for Microsoft Visual Studio Code that includes syntax highlighting and the Solidity compiler.
- **Solidity Visual Studio Code extension, by Nomic Foundation**
  Solidity and Hardhat support by the Hardhat team, including: syntax highlighting, jump to definition, renames, quick fixes and inline solc warnings and errors.
- **Solidity Visual Auditor extension**
  Adds security centric syntax and semantic highlighting to Visual Studio Code.
- **Truffle for VS Code**
  Build, debug and deploy smart contracts on Ethereum and EVM-compatible blockchains. Note: This extension has built-in support for the Truffle Suite which is being sunset. For information on ongoing support, migration options and FAQs, visit the Consensys blog.

3.36.4 Solidity Tools

- **ABI to Solidity interface converter**
  A script for generating contract interfaces from the ABI of a smart contract.
- **abi-to-sol**
  Tool to generate Solidity interface source from a given ABI JSON.
- **Aderyn**
  Command Line Tool that helps find vulnerabilities in Solidity smart contracts. It supports projects built with Hardhat, Foundry, or any custom framework.
- **Doxity**
  Documentation Generator for Solidity.
- **ethdebug**
  A standard debugging data format for smart contracts on Ethereum-compatible networks.
- **Ethlint**
  Linter to identify and fix style and security issues in Solidity.
- **evmdis**
  EVM Disassembler that performs static analysis on the bytecode to provide a higher level of abstraction than raw EVM operations.
- **EVM Lab**
  A collection of tools to interact with the EVM. The package includes a VM, Etherchain API, and a trace-viewer with gas cost display.
- **hevm**
  EVM debugger and symbolic execution engine.
- **leafleth**
  A documentation generator for Solidity smart-contracts.
- **Scaffold-ETH 2**
  Forkable Ethereum development stack focused on fast product iterations.
- **Slippy**
  A simple and powerful linter for Solidity.

- **sol2uml**
  Unified Modeling Language (UML) class diagram generator for Solidity contracts.
- **solc-select**
  A script to quickly switch between Solidity compiler versions.
- **Solidity prettier plugin**
  A Prettier Plugin for Solidity.
- **Solidity REPL**
  Try Solidity instantly with a command-line Solidity console.
- **solgraph**
  Visualize Solidity control flow and highlight potential security vulnerabilities.
- **Solhint**
  Solidity linter that provides security, style guide and best practice rules for smart contract validation.
- **Sourcify**
  Decentralized automated contract verification service and public repository of contract metadata.
- **Sūrya**
  Utility tool for smart contract systems, offering a number of visual outputs and information about the contracts' structure. Also supports querying the function call graph.
- **Universal Mutator**
  A tool for mutation generation, with configurable rules and support for Solidity and Vyper.
- **Wake**
  A Python-based Solidity development and testing framework with built-in vulnerability detectors.

3.36.5 Third-Party Solidity Parsers and Grammars

- **Solidity Parser for JavaScript**
  A Solidity parser for JS built on top of a robust ANTLR4 grammar.

3.37 Contributing

Help is always welcome and there are plenty of options to contribute to Solidity.

In particular, we appreciate support in the following areas:

- Reporting issues.
- Fixing and responding to Solidity's GitHub issues, especially those tagged as "good first issue" which are meant as introductory issues for external contributors.
- Improving the documentation.
- Translating the documentation into more languages.
- Responding to questions from other users on StackExchange and the Solidity Gitter Chat.
- Getting involved in the language design process by proposing language changes or new features in the Solidity forum and providing feedback.

To get started, you can try *Building from Source* in order to familiarize yourself with the components of Solidity and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Solidity.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project — in the issues, pull requests, or Gitter channels — you agree to abide by its terms.

3.37.1 Team Calls

If you have issues or pull requests to discuss, or are interested in hearing what the team and contributors are working on, you can join our public team call:

- Wednesdays at 3PM CET/CEST.

The call takes place on Jitsi.

New topics can be freely added to the agenda and will be scheduled for discussion on the nearest call.

3.37.2 How to Report Issues

To report an issue, please use the GitHub issues tracker. When reporting issues, please mention the following details:

- Solidity version.
- Source code (if applicable).
- Operating system.
- Steps to reproduce the issue.
- Actual vs. expected behavior.

Reducing the source code that caused the issue to a bare minimum is always very helpful, and sometimes even clarifies a misunderstanding.

For technical discussions about language design, a post in the Solidity forum is the correct place (see *Solidity Language Design*).

3.37.3 Workflow for Pull Requests

In order to contribute, please fork off of the `develop` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `develop` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch. This will help us review your change more easily.

Additionally, if you are writing a new feature, please ensure you add appropriate test cases under `test/` (see below).

However, if you are making a larger change, please consult with the Solidity Development Gitter channel (different from the one mentioned above — this one is focused on compiler and language development instead of language usage) first.

New features and bugfixes should be added to the `Changelog.md` file: please follow the style of previous entries, when applicable.

Finally, please make sure you respect the coding style for this project. Also, even though we do CI testing, please test your code and ensure that it builds locally before submitting a pull request.

We highly recommend going through our review checklist before submitting the pull request. We thoroughly review every PR and will help you get it right, but there are many common problems that can be easily avoided, making the review much smoother.

Thank you for your help!

3.37.4 AI-Assisted Contributions

We do not ban the use of AI tools, but we hold all contributions to the same high standard regardless of how they are produced, and we require full transparency about their use.

Submitting AI-generated code means you have reviewed it, understand it, can explain it, and have tested it as thoroughly as if you had written it by hand.

If you used AI tools to generate code, tests, or documentation in any part of your contribution, disclosure in the pull request is mandatory. If we determine that a PR contains undisclosed AI-generated content, we may close it.

3.37.5 Running the Compiler Tests

Prerequisites

For running all compiler tests you may want to optionally install a few dependencies (evmone, z3, Eldarica, *cvc5 <https://github.com/cvc5/cvc5>*).

On macOS systems, some of the testing scripts expect GNU coreutils to be installed. This can be easiest accomplished using Homebrew: `brew install coreutils`.

On Windows systems, make sure that you have a privilege to create symlinks, otherwise several tests may fail. Administrators should have that privilege, but you may also grant it to other users or enable Developer Mode.

Running the Tests

Solidity includes different types of tests, most of them bundled into the Boost C++ Test Framework application `soltest`. Running `build/test/soltest` or its wrapper `scripts/soltest.sh` is sufficient for most changes.

The `./scripts/tests.sh` script executes most Solidity tests automatically, including those bundled into the Boost C++ Test Framework application `soltest` (or its wrapper `scripts/soltest.sh`), as well as command-line tests and compilation tests.

The test system automatically tries to discover the location of the evmone for running the semantic tests.

The `evmone` library must be located in the `deps` or `deps/lib` directory relative to the current working directory, to its parent or its parent's parent. Alternatively, an explicit location for the `evmone` shared object can be specified via the `ETH_EVMONE` environment variable.

`evmone` is needed mainly for running semantic and gas tests. If you do not have it installed, you can skip these tests by passing the `--no-semantic-tests` flag to `scripts/soltest.sh`.

The `evmone` library should end with the file name extension `.so` on Linux, `.dll` on Windows systems and `.dylib` on macOS.

For running SMT tests, the `z3` executable must be present in `PATH`. A few SMT tests use `Eldarica` instead of `z3`. These require its executable (`eld`) to be present in `PATH` for the tests to pass. However, if `Eldarica` is not found, these tests will be automatically skipped.

If `z3` is not present on your system, you should disable the SMT tests by exporting `SMT_FLAGS=--no-smt` before running `./scripts/tests.sh` or running `./scripts/soltest.sh --no-smt`. These tests are `libsolidity/smtCheckerTests`.

> **Note**
>
> To get a list of all unit tests run by Soltest, run `./build/test/soltest --list_content=HRF`.

For quicker results you can run a subset of, or specific tests.

To run a subset of tests, you can use filters: `./scripts/soltest.sh -t TestSuite/TestName`, where `TestName` can be a wildcard `*`.

Or, for example, to run all the tests for the yul disambiguator: `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` has extensive help on all of the options available.

See especially:

- show_progress (-p) to show test completion,
- run_test (-t) to run specific tests cases, and
- report-level (-r) give a more detailed report.

> **Note**
>
> Those working in a Windows environment wanting to run the above basic sets without z3. Using Git Bash, you use: `./build/test/Release/soltest.exe -- --no-smt`. If you are running this in plain Command Prompt, use `.\build\test\Release\soltest.exe -- --no-smt`.

If you want to debug using GDB, make sure you build differently than the "usual". For example, you could run the following command in your `build` folder:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

This creates symbols so that when you debug a test using the `--debug` flag, you have access to functions and variables in which you can break or print with.

The CI runs additional tests (including `solc-js` and testing third party Solidity frameworks) that require compiling the Emscripten target.

Writing and Running Syntax Tests

Syntax tests check that the compiler generates the correct error messages for invalid code and properly accepts valid code. They are stored in individual files inside the `tests/libsolidity/syntaxTests` folder. These files must contain annotations, stating the expected result(s) of the respective test. The test suite compiles and checks them against the given expectations.

For example: `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

A syntax test must contain at least the contract under test itself, followed by the separator `// ----`. The comments that follow the separator are used to describe the expected compiler errors or warnings. The number range denotes the location in the source where the error occurred. If you want the contract to compile without any errors or warning you can leave out the separator and the comments that follow it.

In the above example, the state variable `variable` was declared twice, which is not allowed. This results in a `DeclarationError` stating that the identifier was already declared.

The `isoltest` tool is used for these tests and you can find it under `./build/test/tools/`. It is an interactive tool which allows editing of failing contracts using your preferred text editor. Let's try to break this test by removing the second declaration of `variable`:

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

Running `./build/test/tools/isoltest` again results in a test failure:

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifier already declared.
    Obtained result:
        Success
```

`isoltest` prints the expected result next to the obtained result, and also provides a way to edit, update or skip the current contract file, or quit the application.

It offers several options for failing tests:

- `edit`: `isoltest` tries to open the contract in an editor so you can adjust it. It either uses the editor given on the command-line (as `isoltest --editor /path/to/editor`), in the environment variable `EDITOR` or just `/usr/bin/editor` (in that order).
- `update`: Updates the expectations for contract under test. This updates the annotations by removing unmet expectations and adding missing expectations. The test is then run again.
- `skip`: Skips the execution of this particular test.
- `quit`: Quits `isoltest`.

All of these options apply to the current contract, except `quit` which stops the entire testing process.

Automatically updating the test above changes it to

```
contract test {
    uint256 variable;
}
// ----
```

and re-run the test. It now passes again:

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Note

Choose a name for the contract file that explains what it tests, e.g. `double_variable_declaration.sol`. Do not put more than one contract into a single file, unless you are testing inheritance or cross-contract calls. Each file should test one aspect of your new feature.

Command-line Tests

Our suite of end-to-end command-line tests checks the behaviour of the compiler binary as a whole in various scenarios. These tests are located in test/cmdlineTests/, one per subdirectory, and can be executed using the `cmdlineTests.sh` script.

By default the script runs all available tests. You can also provide one or more file name patterns, in which case only the tests matching at least one pattern will be executed. It is also possible to exclude files matching a specific pattern

by prefixing it with `--exclude`.

By default the script assumes that a `solc` binary is available inside the `build/` subdirectory inside the working copy. If you build the compiler outside of the source tree, you can use the `SOLIDITY_BUILD_DIR` environment variable to specify a different location for the build directory.

Example:

```
export SOLIDITY_BUILD_DIR=~/solidity/build/
test/cmdlineTests.sh "standard_*" "*_yul_*" --exclude "standard_yul_*"
```

The commands above will run tests from directories starting with `test/cmdlineTests/standard_` and subdirectories of `test/cmdlineTests/` that have `_yul_` somewhere in the name, but no test whose name starts with `standard_yul_` will be executed. It will also assume that the file `solidity/build/solc/solc` inside your home directory is the compiler binary (unless you are on Windows – then `solidity/build/solc/Release/solc.exe`).

There are several kinds of command-line tests:

- *Standard JSON test*: contains at least an `input.json` file. In general may contain:
  - `input.json`: input file to be passed to the `--standard-json` option on the command line.
  - `output.json`: expected Standard JSON output.
  - `args`: extra command-line arguments passed to `solc`.
- *CLI test*: contains at least an `input.*` file (other than `input.json`). In general may contain:
  - `input.*`: a single input file, whose name will be supplied to `solc` on the command line. Usually `input.sol` or `input.yul`.
  - `args`: extra command-line arguments passed to `solc`.
  - `stdin`: content to be passed to `solc` via standard input.
  - `output`: expected content of the standard output.
  - `err`: expected content of the standard error output.
  - `exit`: expected exit code. If not provided, zero is expected.
- *Script test*: contains a `test.*` file. In general may contain:
  - `test.*`: a single script to run, usually `test.sh` or `test.py`. The script must be executable.

3.37.6 Running the Fuzzer via AFL

Fuzzing is a technique that runs programs on more or less random inputs to find exceptional execution states (segmentation faults, exceptions, etc). Modern fuzzers are clever and run a directed search inside the input. We have a specialized binary called `solfuzzer` which takes source code as input and fails whenever it encounters an internal compiler error, segmentation fault or similar, but does not fail if e.g., the code contains an error. This way, fuzzing tools can find internal problems in the compiler.

We mainly use AFL for fuzzing. You need to download and install the AFL packages from your repositories (afl, afl-clang) or build them manually. Next, build Solidity (or just the `solfuzzer` binary) with AFL as your compiler:

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

At this stage, you should be able to see a message similar to the following:

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

If the instrumentation messages did not appear, try switching the cmake flags pointing to AFL's clang binaries:

```
# if previously failed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Otherwise, upon execution the fuzzer halts with an error saying binary is not instrumented:

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

Next, you need some example source files. This makes it much easier for the fuzzer to find errors. You can either copy some files from the syntax tests or extract test files from the documentation or the other tests:

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs
```

The AFL documentation states that the corpus (the initial input files) should not be too large. The files themselves should not be larger than 1 kB and there should be at most one input file per functionality, so better start with a small number of. There is also a tool called `afl-cmin` that can trim input files that result in similar behavior of the binary.

Now run the fuzzer (the `-m` extends the size of memory to 60 MB):

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

The fuzzer creates source files that lead to failures in `/tmp/fuzzer_reports`. Often it finds many similar source files that produce the same error. You can use the tool `scripts/uniqueErrors.sh` to filter out the unique errors.

3.37.7 Whiskers

Whiskers is a string templating system similar to Mustache. It is used by the compiler in various places to aid readability, and thus maintainability and verifiability, of the code.

The syntax comes with a substantial difference to Mustache. The template markers {{ and }} are replaced by < and > in order to aid parsing and avoid conflicts with *Yul* (The symbols < and > are invalid in inline assembly, while { and } are used to delimit blocks). Another limitation is that lists are only resolved one depth and they do not recurse. This may change in the future.

A rough specification is the following:

Any occurrence of `<name>` is replaced by the string-value of the supplied variable `name` without any escaping and without iterated replacements. An area can be delimited by `<#name>...</name>`. It is replaced by as many concatenations of its contents as there were sets of variables supplied to the template system, each time replacing any `<inner>` items by their respective value. Top-level variables can also be used inside such areas.

There are also conditionals of the form `<?name>...<!name>...</name>`, where template replacements continue recursively either in the first or the second segment depending on the value of the boolean parameter `name`. If `<?+name>...<!+name>...</+name>` is used, then the check is whether the string parameter `name` is non-empty.

3.37.8 Documentation Style Guide

In the following section you find style recommendations specifically focusing on documentation contributions to Solidity.

English Language

Use International English, unless using project or brand names. Try to reduce the usage of local slang and references, making your language as clear to all readers as possible. Below are some references to help:

- Simplified technical English
- International English

> **Note**
>
> While the official Solidity documentation is written in English, there are community contributed *Translations* in other languages available. Please refer to the translation guide for information on how to contribute to the community translations.

Title Case for Headings

Use title case for headings. This means capitalise all principal words in titles, but not articles, conjunctions, and prepositions unless they start the title.

For example, the following are all correct:

- Title Case for Headings.
- For Headings Use Title Case.
- Local and State Variable Names.
- Order of Layout.

Expand Contractions

Use expanded contractions for words, for example:

- "Do not" instead of "Don't".
- "Can not" instead of "Can't".

Active and Passive Voice

Active voice is typically recommended for tutorial style documentation as it helps the reader understand who or what is performing a task. However, as the Solidity documentation is a mixture of tutorials and reference content, passive voice is sometimes more applicable.

As a summary:

- Use passive voice for technical reference, for example language definition and internals of the Ethereum VM.
- Use active voice when describing recommendations on how to apply an aspect of Solidity.

For example, the below is in passive voice as it specifies an aspect of Solidity:

> Functions can be declared `pure` in which case they promise not to read from or modify the state.

For example, the below is in active voice as it discusses an application of Solidity:

> When invoking the compiler, you can specify how to discover the first element of a path, and also path prefix remappings.

Common Terms

- "Function parameters" and "return variables", not input and output parameters.

Code Examples

A CI process tests all code block formatted code examples that begin with `pragma solidity`, `contract`, `library` or `interface` using the `./test/cmdlineTests.sh` script when you create a PR. If you are adding new code examples, ensure they work and pass tests before creating the PR.

Ensure that all code examples begin with a `pragma` version that spans the largest where the contract code is valid. For example `pragma solidity >=0.4.0 <0.9.0;`.

Running Documentation Tests

Make sure your contributions pass our documentation tests by running `./docs/docs.sh` that installs dependencies needed for documentation and checks for any problems such as broken links or syntax issues.

3.37.9 Solidity Language Design

To actively get involved in the language design process and to share your ideas concerning the future of Solidity, please join the Solidity forum.

The Solidity forum serves as the place to propose and discuss new language features and their implementation in the early stages of ideation or modifications of existing features.

As soon as proposals get more tangible, their implementation will also be discussed in the Solidity GitHub repository in the form of issues.

In addition to the forum and issue discussions, we regularly host language design discussion calls in which selected topics, issues or feature implementations are debated in detail. The invitation to those calls is shared via the forum.

We are also sharing feedback surveys and other content that is relevant to language design in the forum.

If you want to know where the team is standing in terms of implementing new features, you can follow the implementation status in the Solidity GitHub project. Issues in the design backlog need further specification and will either be discussed in a language design call or in a regular team call. You can see the upcoming changes for the next breaking release by changing from the default branch (*develop*) to the breaking branch.

For ad-hoc cases and questions, you can reach out to us via the Solidity-dev Gitter channel — a dedicated chatroom for conversations around the Solidity compiler and language development.

We are happy to hear your thoughts on how we can improve the language design process to be even more collaborative and transparent.

3.38 Language Influences

Solidity is a curly-bracket language that has been influenced and inspired by several well-known programming languages.

Solidity is most profoundly influenced by C++, but also borrowed concepts from languages like Python, JavaScript, and others.

The influence from C++ can be seen in the syntax for variable declarations, for loops, the concept of overloading functions, implicit and explicit type conversions and many other details.

In the early days of the language, Solidity used to be partly influenced by JavaScript. This was due to function-level scoping of variables and the use of the keyword `var`. The JavaScript influence was reduced starting from version 0.4.0. Now, the main remaining similarity to JavaScript is that functions are defined using the keyword `function`. Solidity also supports import syntax and semantics that are similar to those available in JavaScript. Besides those points, Solidity looks like most other curly-bracket languages and has no major JavaScript influence anymore.

Another influence to Solidity was Python. Solidity's modifiers were added trying to model Python's decorators with a much more restricted functionality. Furthermore, multiple inheritance, C3 linearization, and the `super` keyword are taken from Python as well as the general assignment and copy semantics of value and reference types.

3.39 Solidity Brand Guide

This brand guide features information on Solidity's brand policy and logo usage guidelines.

3.39.1 The Solidity Brand

The Solidity programming language is an open-source, community project governed by a core team. The core team is sponsored by the Ethereum Foundation.

This document aims to provide information about how to best use the Solidity brand name and logo.

We encourage you to read this document carefully before using the brand name or the logo. Your cooperation is highly appreciated!

3.39.2 Solidity Brand Name

"Solidity" should be used to refer to the Solidity programming language solely.

Please do not use "Solidity":

- To refer to any other programming language.
- In a way that is misleading or may imply association of unrelated modules, tools, documentation, or other resources with the Solidity programming language.
- In ways that confuse the community as to whether the Solidity programming language is open-source and free to use.

3.39.3 Solidity Logo License

The Solidity logo is distributed and licensed under a Creative Commons Attribution 4.0 International License.

This is the most permissive Creative Commons license and allows reuse and modifications for any purpose.

You are free to:

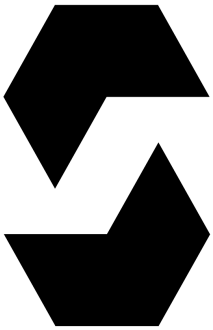
- **Share** — Copy and redistribute the material in any medium or format.
- **Adapt** — Remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests that the Solidity core team endorses you or your use.

When using the Solidity logo, please respect the Solidity logo guidelines.

3.39.4 Solidity Logo Guidelines



(Right click on the logo to download it.)

Please do not:

- Change the ratio of the logo (do not stretch it or cut it).
- Change the colors of the logo, unless it is absolutely necessary.

3.39.5 Credits

This document was, in parts, derived from the Python Software Foundation Trademark Usage Policy and the Rust Media Guide.

INDEX

Symbols

--allow-paths, 199, **369**
--base-path, 199, 200, **367**
--experimental, **214**
--include-path, 199, **367**
--libraries, **199**
--link, **199**
--no-import-callback, **363**
--standard-json, 200, **202**
<stdin>, 365

A

abi, 93, 95, 261
    decode, 167
    encode, 167
    encodeCall, 167
    encodePacked, 167
    encodeWithSelector, 167
    encodeWithSignature, 167
ABI coder, **47**
abstract contract, 148, **150**
access
    restricting, 396
account, **11**
addmod, 96, 169
address, 11, 56, 61
    balance, 168
    code, 168
    codehash, 168
    send, 168
    transfer, 168
allowed paths, 199, **369**
analyse, 214
anonymous, 170
application binary interface, 261
array, 71, **72**, 128
    allocating, **74**
    dangling storage references, **79**
    length, **76**
    pop, **76**
    push, **76**
    slice, **80**
array of strings, 128
asm, **160**, 214, **342**
assembly, **160**, **342**
assemblyFlags (*SolidityParser production rule*), 185
assemblyStatement (*SolidityParser production rule*), 185
assert, 95, **109**, 169
assignment, 88, **105**
    destructuring, **105**
auction
    blind, 19
    open, 19

B

balance, 11, 56, 97
ballot, 16
base
    constructor, **148**
base class, **140**
base path, 199, **367**
base slot, **126**
blind auction, 19
blobhash, 168
block, **11**, 93, 168
    basefee, 93, 168
    blobbasefee, 93, 168
    chainid, 168
    coinbase, 168
    difficulty, 168
    gaslimit, 168
    number, 93, 168
    prevrandao, 168
    timestamp, 93, 168
block (*SolidityParser production rule*), 183
blockhash, 168
bool, **53**
booleanLiteral (*SolidityParser production rule*), 182
break, 100
breakStatement (*SolidityParser production rule*), 185
Bugs, 282
byte array, 60
bytes, 63, **73**

concat, 167
bytes members, 95
bytes32, 60
bytes-concat, **73**

C

C3 linearization, **148**
call, 56, 97
callArgumentList (*SolidityParser production rule*), 173
callcode, 97, 152
cast, **89**
catchClause (*SolidityParser production rule*), 185
checked, 107
cleanup, 222
codehash, 97
coding style, 373
coin, 10
coinbase, 93
commandline compiler, **198**
comment, **49**
common subexpression elimination, 236
compile target, 200
compiler
    commandline, 198
compound operators, **88**
constant, **124**, 170
constant propagation, 236
constantVariableDeclaration (*SolidityParser production rule*), 177
constructor, 114, **147**
    arguments, 114
constructorDefinition (*SolidityParser production rule*), 174
continue, 100
continueStatement (*SolidityParser production rule*), 185
contract, 50, **114**
    abstract, 148, **150**
    base, **140**
    creation, **114**
    interface, **151**
    modular, 37
    precompiled, **15**
contract creation, 14
contract type, **60**
contract verification, 256
contractBodyElement (*SolidityParser production rule*), 172
contractDefinition (*SolidityParser production rule*), 171
contracts
    creating, 103
creationCode, 99
cryptography, 96, 169
custom storage layout, **126**
custom type, 65

D

data, 93
dataLocation (*SolidityParser production rule*), 179
days, **92**
deactivate, 14
DecimalNumber (*SolidityLexer production rule*), 196
declarations, 106
default value, 106
delegatecall, 14, 56, 97, 152
delete, **88**
denomination, **92**
    ether, **92**
    time, **92**
deriving, **140**
difficulty, 93
direct import, **365**
dirty bits, 222
do/while, 100
DoubleQuotedPrintable (*SolidityLexer production rule*), 195
doWhileStatement (*SolidityParser production rule*), 184
dynamic array, 128

E

ecrecover, 96, 169
elementaryTypeName (*SolidityParser production rule*), 179
else, 100
emitStatement (*SolidityParser production rule*), 185
EmptyStringLiteral (*SolidityLexer production rule*), 195
encode, 93
encoding, 95
enum, 50, 64
enumDefinition (*SolidityParser production rule*), 176
erc7201, 96, 169
error, **138**, 270
errorDefinition (*SolidityParser production rule*), 177
errorParameter (*SolidityParser production rule*), 177
errors, **109**
EscapeSequence (*SolidityLexer production rule*), 195
escrow, 25
ether, **92**
ethereum virtual machine, **11**
evaluation order
    expression, **220**
    function arguments, **221**
event, 9, 50, **136**
    anonymous, **136**

indexed, **136**
topic, **136**
eventDefinition (*SolidityParser production rule*), 177
eventParameter (*SolidityParser production rule*), 177
evm, **11**
EVM version, **200**
evmasm, **160**, **342**
exception, **109**
Experimental mode, **214**
expression (*SolidityParser production rule*), 180
expressionStatement (*SolidityParser production rule*), 186
external, 116, 169

F

fallback function, **132**
fallbackFunctionDefinition (*SolidityParser production rule*), 175
false, **53**
file://, 372
filesystem path, 49
finney, **92**
fixed, **56**
fixed point number, **56**
FixedBytes (*SolidityLexer production rule*), 189
for, 100
forStatement (*SolidityParser production rule*), 184
function, 50
call, 14, **100**
external, 100
fallback, 132
free, **127**, 156
getter, **118**
internal, 100
modifier, 50, **120**, 396, 398
pure, 130
receive, 131
view, 129
function parameter, 100
function pointers, 222
function type, **66**
functionDefinition (*SolidityParser production rule*), 174
functions, **127**
functionTypeName (*SolidityParser production rule*), 179

G

gas, **12**, 93
gas price, **12**, 93
gasleft, 168
getter
function, **118**
goto, 100
gwei, **92**

H

HexNumber (*SolidityLexer production rule*), 196
HexString (*SolidityLexer production rule*), 195
hexStringLiteral (*SolidityParser production rule*), 183
Host Filesystem Loader, **363**
hours, **92**

I

Identifier (*SolidityLexer production rule*), 196
identifier (*SolidityParser production rule*), 182
identifierPath (*SolidityParser production rule*), 173
if, 100
ifStatement (*SolidityParser production rule*), 184
import, **49**
direct, 365
path, 49, **365**
relative, **366**
remapping, **370**
import callback, 49, **363**
importDirective (*SolidityParser production rule*), 171
include paths, 199, **367**
indexed, 170
inheritance, **140**
multiple, **148**
inheritance list, 148
inheritanceSpecifier (*SolidityParser production rule*), 172
inline
arrays, **74**
inlineArrayExpression (*SolidityParser production rule*), 182
installing, **38**
instruction, **13**
int, **53**
integer, **53**, 61
interface contract, **151**
interfaceDefinition (*SolidityParser production rule*), 172
internal, 116, 169
iterable mappings, **85**
iulia, 342

J

julia, 342

K

keccak256, 96, 169

L

layout at, **126**

length, 76
library, 14, **152**, 156
libraryDefinition (*SolidityParser production rule*), 172
license, **47**
linearization, **148**
linker, **199**
literal
    address, **61**, 92
    array, **74**
    conversion, **91**
    hexadecimal, **63**, 91
    hexadecimal number, 91
    in Yul, **344**
    rational, **61**, 91
    string, **62**, 91
    unicode, **63**
literal (*SolidityParser production rule*), 182
literalWithSubDenomination (*SolidityParser production rule*), 182
location, 71
log, 14
lvalue, 88

M

mapping, 9, **83**, 223
mappingKeyType (*SolidityParser production rule*), 186
mappingType (*SolidityParser production rule*), 186
memory, **12**, 71
message call, **14**
metadata, 256
minutes, **92**
modifierDefinition (*SolidityParser production rule*), 175
modifierInvocation (*SolidityParser production rule*), 173
modifiers, 170
modular contract, 37
module, 49
msg, 93
    data, 168
    sender, 168
    sig, 168
    value, 168
mulmod, 96, 169

N

natspec, 49
new, 74, **103**
NonEmptyStringLiteral (*SolidityLexer production rule*), 194
number, 93
numberLiteral (*SolidityParser production rule*), 183

O

open auction, 19
operator, **87**
    precedence, **89**, 166
    user-defined, **156**
optimiser, 236
optimizer, 236
origin, 93
overload, **134**
overrideSpecifier (*SolidityParser production rule*), 174
overriding
    function, **144**
    modifier, **146**

P

packed, 95
parameter, **100**
    function, 100
    input, 100
    output, 100
parameterList (*SolidityParser production rule*), 174
path (*SolidityParser production rule*), 171
payable, 170
pop, 76
pragma, **47**
    abicoder, **47**
    ABIEncoderV2, 47, **48**
    experimental, **48**
    SMTChecker, **48**
    version, **47**
PragmaToken (*SolidityLexer production rule*), 198
precompiled contracts, **15**
precompiles, **15**
prevrandao, 93
private, 116, 169
public, 116, 169
purchase, 25
pure, 170
pure function, **130**
push, 76

R

rational number, 61
receive, **131**
receive ether function, **131**
receiveFunctionDefinition (*SolidityParser production rule*), 176
reference type, **71**
relative import, **366**
remapping
    context, **370**
    import, **370**

    prefix, **370**
    target, 369, **370**
Remix IDE, 49, 372
remote purchase, 25
require, 95, **109**, 138, 169
return, 100
return array, 128
return string, 128
return struct, 128
return variable, 100
returnStatement (*SolidityParser production rule*), 185
revert, 95, **109**, 138, 169
revertStatement (*SolidityParser production rule*), 185
ripemd160, 96, 169
runtimeCode, 99

S

safe math, **107**
safemath, 107
scoping, **106**
seconds, **92**
selector
    of a function, 162, **261**
    of a library function, **155**
    of an error, **138**, 270
    of an event, **137**
selfdestruct, **14**, 98, 169
send, 56, 97
sender, 93
set, 153
sha256, 96, 169
SignedIntegerType (*SolidityLexer production rule*), 191
SingleQuotedPrintable (*SolidityLexer production rule*), 195
solc, **198**
SolidityLexer (*grammar*), 189
SolidityParser (*grammar*), 170
source file, 49
source mappings, 235
source unit, 49
source unit name, 49, **363**
sourceUnit (*SolidityParser production rule*), 170
spdx, 47
stack, **12**
standard input, 365
standard JSON, **202**, 364
state machine, 398
state variable, 50, 223
statement (*SolidityParser production rule*), 183
stateMutability (*SolidityParser production rule*), 174
stateVariableDeclaration (*SolidityParser production rule*), 177
staticcall, 56, 97
stdin, 365
storage, 11, **12**, 71, 223
storage layout specifier, **126**
string, 62, **73**, 128
    concat, 167
string members, 95
string-concat, **73**
stringLiteral (*SolidityParser production rule*), 182
struct, 50, 71, **81**, 128
structDefinition (*SolidityParser production rule*), 176
structMember (*SolidityParser production rule*), 176
style, 373
subcurrency, **8**
SubDenomination (*SolidityLexer production rule*), 190
super, 98, 169
switch, 100
symbolAliases (*SolidityParser production rule*), 171
szabo, **92**

T

this, 98, 169
throw, **109**
timestamp, 93
tload, 122
transaction, 10, **12**
transfer, 56, 97
transient, **122**
transient storage, **12**, **122**, 223
true, **53**
tryStatement (*SolidityParser production rule*), 185
tstore, 122
tupleExpression (*SolidityParser production rule*), 181
tx
    gasprice, 168
    origin, 168
type, 53, 99
    contract, **60**
    conversion, **89**
    creationCode, 169
    function, **66**
    interfaceId, 169
    max, 169
    min, 169
    name, 169
    reference, **71**
    runtimeCode, 169
    struct, **81**
    value, **53**
typeName (*SolidityParser production rule*), 179

U

ufixed, **56**
uint, **53**

unchecked, 107
uncheckedBlock (*SolidityParser production rule*), 183
UnicodeStringLiteral (*SolidityLexer production rule*), 195
unicodeStringLiteral (*SolidityParser production rule*), 183
UnsignedIntegerType (*SolidityLexer production rule*), 192
UnusedStoreEliminator, **251**
user defined value type, **65**
userDefinableOperator (*SolidityParser production rule*), 178
userDefinedValueTypeDefinition (*SolidityParser production rule*), 176
using for, 153, **156**
usingAliases (*SolidityParser production rule*), 178
usingDirective (*SolidityParser production rule*), 178

V

value, 93
value type, **53**
variable
    return, 100
variableDeclaration (*SolidityParser production rule*), 179
variableDeclarationStatement (*SolidityParser production rule*), 186
variableDeclarationTuple (*SolidityParser production rule*), 186
variably sized array, 128
VFS, **363**
view, 170
view function, **129**
virtual filesystem, 49, **363**
visibility, **116**, 169
visibility (*SolidityParser production rule*), 173
voting, 16

W

weeks, **92**
wei, **92**
while, 100
whileStatement (*SolidityParser production rule*), 184
withdrawal, 394

Y

years, **92**
yul, **342**
yulAssignment (*SolidityParser production rule*), 187
yulBlock (*SolidityParser production rule*), 187
yulBoolean (*SolidityParser production rule*), 188
YulDecimalNumber (*SolidityLexer production rule*), 198
YulEVMBuiltin (*SolidityLexer production rule*), 196
yulExpression (*SolidityParser production rule*), 189
yulForStatement (*SolidityParser production rule*), 187
yulFunctionCall (*SolidityParser production rule*), 188
yulFunctionDefinition (*SolidityParser production rule*), 188
YulHexNumber (*SolidityLexer production rule*), 198
YulIdentifier (*SolidityLexer production rule*), 197
yulIfStatement (*SolidityParser production rule*), 187
yulLiteral (*SolidityParser production rule*), 188
yulPath (*SolidityParser production rule*), 188
yulStatement (*SolidityParser production rule*), 186
YulStringLiteral (*SolidityLexer production rule*), 198
yulSwitchStatement (*SolidityParser production rule*), 187
yulVariableDeclaration (*SolidityParser production rule*), 187